Texas Engineering Experiment Station
The Texas A&M University System
College Station, Texas 77843

Hypercubes for Critical Spacecraft Command Verification

Final Report

Vidya Gummadi & Arkady Kanevsky
Department of Computer Science
Texas A&M University
College Station, TX 77843

October 2, 1992

Abstract

A spacecraft can be modeled as a collection of states, with certain actions (commands) leading the spacecraft from one state to another. Here, we present the outline of a technique to generate models from the existing sequence checking code in a semi-automated way. To further speedup the simulation on the hypercube, a technique for Parallel Discrete Event Simulation (PDES) based upon the prediction techniques is also presented. The prediction technique is introduced with simulation results obtained on the Ncube and the performance of the proposed technique is compared to the Time Warp algorithm.

1 Introduction

Spacecrafts are controlled by sequences of time-tagged control commands which are essentially onboard computer programs. The sequence checking employed (presently) is inherently sequential and is implemented partly in PL/1 and partly in, what can be referred to as, the assembly code of a spacecraft. A sequence of control commands needs to be verified for correct execution before the commands are executed on the spacecraft to avoid the catastrophic effects of incorrect execution. Thus, the sequences must be correct and must not have unexpected side effects. One of the keystones of this process is the *checker* program which takes as input the sequence to be sent to the spacecraft. The checker consists of event logic (about 200 pages of code) and several dozens of extra routines for handling models, I/O, interfaces and some other modules of the checker including stack implementation and parsers. The main objective of this work is to replace this existing event-logic algorithm with a state-based algorithm (that we are developing) to improve the performance of the checker.

Spacecraft command sequence checking consists of two parts: *system specification* and *system verification*. System specification can be further divided into two parts: the first part consisting of functional and behavioral models of the spacecraft on-board computer, engineering system, and all related subsystems; and the second part consists of a set of mission specific flight rules or constraints, that restrict the allowable set of parameter values in the model. System verification is an attempt to increase the user's confidence that a specific set of programs satisfy the functional and behavioral aspects of the model without violating the flight rules.

This report was prepared for the Jet Propulsion Laboratory, California Institute of Technology, sponsored by the National Aeronautics and Space Administration.

In the past, command sequence checking for unmanned spacecrafts was based on software technology developed for the Voyager in the mid 70's. The method in which checker was implemented is through indirect function calls for every variable (node) of the code. There are a set of variables for each command of the spacecraft. For every variable X, there is a dependency table that represents all the variables that can be changed if the variable X is changed. The checker takes one command at a time and sets the variable for this command to *true* and puts all of its dependents on the stack. Then it removes one variable at a time from the stack and checks if it needs to be changed. If it is changed, then its dependents need to be put onto the stack and so on. When the stack is empty, the command variable is set to *false* and the process is repeated. This emulates the electric pulse of the command on a spacecraft. So, this sequence checker is optimized for sequential execution on the Univac mainframe machine, and a special-purpose event-logic language is used to specify both the engineering model and the flight rules.

Each command consists of several parameters. They represent the name of the command, the computer on which the command is to be run, the time at which the command should be executed, and some others, including the input parameters to the command. Some of the commands create *new* commands that need to be executed at some deterministic time in the future with respect to the current command. The new commands are called *delay commands* and represent the actions the spacecraft will take upon receiving the command. These commands represent the single hardest problem to parallel implementation of Parallel Checker which makes use of lookahead techniques. In the proposed technique where we use Parallel Discrete Event Simulation for constraint checking, we do not consider delay commands(at present).

The limitations of the current implementations that motivated the development of the proposed methodology are described here.

- The event-logic language has not evolved enough to take advantage of advances in software engineering technology like modular and structured programming, strong typing and data abstractions.
- The language and programming environment used for specifying the engineering model and flight rules, are not based on a well founded formalism. This implies that even if the existing checker code does not report an error, one cannot claim that a flight rule was not violated.
- The event-logic language is inherently sequential, and is very difficult to parallelize. As more complex systems are defined, the number of flight rules become numerous which would result in unacceptably long verification sessions. Thus, a new approach to specification and verification must be explored to allow for parallelism.

We now briefly describe how our work addresses each of the above issues.

The constraint specification methodology developed at JPL separates the models of how the spacecraft works from the models of the constraints on how the spacecraft is flown. The spacecraft is modeled as a collection of states, with certain actions (commands) leading the spacecraft from one state to another. The current state represents the state of some device on the spacecraft such as: gyroscope - ON, heater A - OFF etc. The set of spacecraft commands is responsible for transitions among the various states. The spacecraft is basically modeled as a set of finite state machines, each one corresponding to a particular device of the spacecraft with some constraint associated with it. The finite state machines (models) could be interdependent, thus, a transition in one model might cause some other transition in another model. As a result, the interdependencies among the various models need to be kept track of.

This new specification will allow for improvement in sequence checking in several specific areas. The new representation will assist in the development and encoding of the flight rules and ground and spacecraft constraints since it will now be easier to add rules to the code. The new specification provides insight into the changes in the state of the spacecraft implied by the issuance of a given spacecraft command (or set of commands) by making clear the interrelationships among various commands and subsystems implied by a spacecraft constraint such as a flight rule. It also provides insight into the (frequently non-obvious) other flight rules implied by a given flight rule.

The data structure of a model contains the state values of the model, the commands that cause transition from one state to another, and the interdependencies among the models (the side-effects caused by a transition in a model). Thus, each model has complete information necessary to verify the correctness of a sequence of commands.

One of the problems of using finite state machines as models for the spacecraft is the exponential explosion in the number of states. Action tables are used to denote the set of activities that take place when a machine makes a transition from one state to another.

Constraints are specified using rule-based clauses typical of logic programming languages. The clauses are used to specify that a flight rule must be checked on each occurrence of a specific event in the engineering model. The system specification and verification model developed at JPL consists of two layers. At the bottom layer lies the system specification and simulation environment. This consists of a number of state variables as well as data structures that describe the spacecraft. Given a command sequence, the execution of the sequence is simulated after specification and thus various aspects of program behavior such as termination, deadlocks etc. are verified. If we consider the system to be a finite automata, then simulating program execution is similar to verifying whether the input string of commands is part of the language accepted by the automata. This does not guarantee that the program is correct or that it violates some flight rule. The second layer in the system specification and verification model consists of a constraint checker which verifies whether the system satisfies specific flight rules. These checks are triggered by specific events that occur at the simulation layer.

The implementation details of the parallel constraint checking follow. All the spacecraft models are distributed over the multiple processors of the parallel computer. Various heuristics could be employed to determine which model needs to be placed on which processor; but the best method would be to place all interdependent models on a single processor in order to reduce the communication between models. The communication between models is assumed to be performed using shared state variables. The time-ordered input sequence of commands that have to be simulated and verified for correct execution are partitioned statically according to the model(s) to which they belong. These partial sequences are then checked in parallel. When the partial sequences on each processor affect each other, a synchronization mechanism similar to simplified discrete event simulation takes over and makes sure that these interactions are dealt with correctly. At present it is not clear how commands that are shared among several models are allocated on the different processors.

Some preliminary tests (using three models on three processors) have been run at JPL which show that the simulation part of the algorithm runs about 1.7 times faster on three processors rather than on a single processor. Work is also ongoing to develop a semi-automated way to read the Univac Galileo constraint and model database and convert it to the required notation (in the form of models). Below, we present a technique which might be feasible for automatically generating models from the sequence checking code.

2 Outline of Semi-automated Model Generator

In this section, we present the outline of the algorithm that was proposed to semi-automatically generate the models from the Univac Galileo constraint and model database. The inputs necessary for the proposed algorithm are the various *commands* of the spacecraft, the actual code for sequence checking - the *checker-code*, and a backward-propagation file *seqgen-back* for all the variables used in the code. The backward propagation file generates a list of all the commands which could possibly change the value of a particular variable. This backward propagation file is necessary to automate the transition from an old to a new representation i.e. the model representation. The output of the algorithm is the set of models generated from the code.

The model of a spacecraft is basically a collection of states with transitions among the different states. The set of spacecraft commands is responsible for transitions among the various states of a model, so, the effect of all the possible commands of a spacecraft need to be considered. The proposed algorithm is presented below.

For each command cmd_i in *commands*, do the following:

1. Assign initial values to the variables required for cmd_i, and trace the computations involving the variables of cmd_i in the *checker-code*. Notice that the command cmd_i itself is a variable. We compute a list of all the variables whose final values are different from their initial values (assigned before execution of cmd_i).

2. For every variable var_i in the changed-variables list, we find the commands which could change the variable in the seqgen-back file. The models that are built using the changed-variable list are in actuality finite state machines with different commands as transitions from one state to the other. The commands computed are commands which either cause transitions from one state of the model ($model_i$) to the other or other commands which could possibly trigger $model_i$.

   Note: It is quite possible that most of the models we are trying to generate are simple two or three state machines with a set of commands with proper inputs taking the machine from one state to the other and another set of commands taking the machine back to the initial state. We should be able to determine whether the model is two state or more by keeping track of the values of the variables that are being changed. For example if at one instance we have var_i = X and after the computation we have var_j = NOT X then it is quite possible that we can assume that the model involving the variable X would be a two state machine.

3. In this step, we need to implement the model obtained in step 2 using C. The models that have been manually implemented are simple with the transitions between states being represented by changing the value of the variable corresponding to the model to true or false (corresponding to turning a particular device ON or OFF respectively) at an appropriate time. The actual method that we propose to use for conversion of the models into C is outlined in the next section.

3 Implementation of the models using C

We propose to develop a method by which we can automatically convert the models into code in C having knowledge of the variables involved and the times of the transitions. It seems to be simple for two state or three state machines. We could have functions for two-state and three-state models such

that given the variable names and the times of transitions we can generate the models by printing out the code with the appropriate variable names and times in an output file. The assumption made here is that a two-state machine will have not more than one variable being changed and a three-state machine has no more than two variables being changed (which seems to be the case in the models generated manually).

There are instances where one model might trigger another model either through a delay or directly, as in the case of the gyro and accelerometer models. Both models trigger the inertial sensor model and depending on the values returned by the inertial sensor model, they perform corresponding actions. Such models might be merged together by having transitions (which are activated conditionally) from one model to the other. Another issue of concern is the case where a lot of models change a particular variable - in which case the different models can be merged to form a single model. It is extremely important to be able to develop a method to merge the models together while retaining the consistency of individual models.

In this approach we have assumed that we have the same model for different devices of the same type i.e. device A, device B etc. In the case of some devices there are restrictions such as *both devices should not be ON simultaneously* which leads to a forbidden state in the finite state machine. Therefore, we need to incorporate conditions so that when one device is ON the other can never be turned ON.

By using the above mentioned algorithm, most of the models can be generated even though not all of them. In Appendix A, we present some of the models that were generated using the above mentioned algorithm. Since the specification of flight rules and constraints for all the spacecrafts is not in the same assembly language, we would have to consider each spacecraft's code individually to find a method of generating the models. Therefore, the obvious disadvantage of this method is that this algorithm may not be applicable to all the spacecrafts.

For a deterministic application (such as sequence checking where we can deterministically state the interdependencies among the various models generated), the above method of parallel checking would be efficient but in non-deterministic applications the above method would not be as efficient. Below, we present a more general approach for sequence checking of events.

4 Constraint Checking using Parallel Discrete Event Simulation

An important reason why spacecraft sequencing works is that the spacecraft systems have evolved to be as deterministic as possible. A given spacecraft command is expected to always have a deterministic effect, and the spacecraft's state is periodically reported down to the ground so that any possible inconsistency can be identified. This determinism has some interesting applications for the parallel implementation of the sequence checking software. This software resembles parallel discrete event simulation. If one has fully deterministic knowledge of all communications, then an interesting special-purpose, conservative method can be developed for the application [2, 13]. In the case of applications where one does not have fully deterministic knowledge of all possible communication involved, in the following section, we present a technique for efficient sequence checking.

Parallel (distributed) discrete event simulation refers to the execution of a single discrete event simulation program on a parallel computer. A discrete event simulation model assumes that the system being simulated changes state only at discrete points in simulated time. In other words the simulation

is event driven in the sense that the simulation model changes from one state to another only on the occurrence of an event.

A simulation program can be executed on a parallel computer by decomposing the simulation application into a set of concurrently executing processes [8]. Other approaches for exploiting parallelism are : using dedicated functional units to implement specific *sequential* simulation functions; using a hierarchical decomposition of the simulation model to allow an event consisting of several subevents to be processed concurrently; and to execute independent, sequential simulation programs on different processors (termed as replicated trials approach). Due to the range of applications of PDES, a considerable amount of work has been done in the past and the research in this area continues today where new approaches or enhancements to existing approaches are being developed.

Sequential simulators typically utilize three data structures:

1. the *global state variables* that describe the state of the system,
2. an *event list* containing all pending (and previous) events that have to be scheduled, but have not yet taken effect, and
3. a *global clock* variable to denote how far the simulation has progressed.

Each event contains a timestamp, and usually denotes some change in the state of the system being simulated. The timestamp indicates when change occurs in the actual system. The *main loop* of the simulator repeatedly removes the smallest timestamped event from the event list, and processes the event. Processing an event involves executing some simulator code to effect the appropriate change in state, and scheduling zero or more new events into the simulated future in order to model causality relationships in the system under investigation.

In terms of the spacecraft, we can consider an event to correspond to a profile of activity, such as turning the accelerometer ON, etc. In the present simulation, not much importance is placed on what an event actually does, rather we are more concerned with the time required for the execution of an event. Sequence commands must be in time order for the computation to be correct. In other words, it is critical to select the smallest timestamped event from the event list to be processed, since there exists a possibility where a higher timestamped event may modify the state variables that are used by the lower timestamped event. This would amount to simulating a system in which the future could affect the past which is obviously unacceptable. Such errors are referred to as *causality errors*. Most of the existing strategies in order to avoid this problem simply forbid the direct access to shared (global) variables. The system that is being simulated is the *physical system* and it is composed of several *physical processes* that interact at various points in simulated time. All interactions between physical processes are modeled as timestamped event messages between logical processes. One can ensure the avoidance of the causality errors by following the following rule:

- *Local Causality Constraint* - A discrete event simulation, consisting of logical processes (LPs) that interact exclusively by exchanging timestamped messages, obeys the local causality constraint if and only if each LP processes events in nondecreasing timestamp order.

Adherence to this constraint is sufficient, though always not necessary, to guarantee that no causality errors occur. Note that this rule is in effect not only for the original scheduled discrete event but also for the new events that are scheduled through the execution of the original events and are put on the event list with the appropriate timestamp in the future. The major problems with parallel discrete event

simulation are *load balancing* and *communication*. These factors maximize the speed-up and ensure correctness of the simulation.

PDES mechanisms broadly fall into two categories *conservative* and *optimistic*. A lot of different variations of these techniques with applications can be found in the survey article [8]. Conservative approaches like Chandy-Misra [7, 5], strictly avoid the possibility of any causality error ever occurring. These approaches rely on some strategy to determine when it is safe to process an event (they must determine when all events that could affect the event in question have been processed). On the other hand, optimistic approaches such as the Time-Warp mechanism [4, 14], use a *detection and recovery* approach: causality errors are detected, and a *rollback* mechanism is invoked to recover.

4.1 Conservative and Optimistic Approaches

Conservative Mechanisms:

The basic problem conservative mechanisms must solve is determining when it is safe to process an event. More precisely, if a process contains an unprocessed event $Event_i$ with a timestamp T_i(and no other with a smaller timestamp), and that event (process) can determine that it is impossible for it to later receive another event with timestamp less than T_i, then the process can safely process $Event_i$ since it can guarantee that doing so will not later result in a violation of the local causality constraint [7, 5].

Lookahead is the ability to predict what will happen, or what will not happen in an event's simulated future [8]. If a process at simulated time *Clock* can predict with complete certainty all events it will generate upto simulated time $Clock + L$, the process is said to have a lookahead of L. Lookahead enhances one's ability to predict future events, which in turn, can be used to determine which other events are safe to process.

Conservative approaches are most often overly pessimistic, and force sequential execution when it is not necessary. If there were no lookahead, the smallest timestamped event in the simulation *could* affect other pending events, forcing sequential execution no matter what conservative protocol is used. Conservative algorithms appear to be poorly suited for simulating applications with poor lookahead properties, even if there is a healthy amount of parallelism available.

Optimistic Mechanisms:

Optimistic methods detect and recover from causality errors; they do not strictly avoid them. In contrast to conservative mechanisms, optimistic strategies need not determine when it is safe to proceed; instead they determine when an error has occurred, and invoke a procedure to recover [8]. One advantage of this approach is that it allows the simulator to exploit parallelism situations where it is possible causality errors might occur, but in fact do not. Also, dynamic creation of logical processes can be easily accommodated.

The Time-Warp mechanism [14, 4] is the most well known optimistic protocol. In this approach, a causality error is detected whenever an event message is received that contains a timestamp smaller than that of the process clock which is the timestamp of the last processed message. Recovery is accomplished by undoing the effects of all events that have been processed prematurely by the process receiving the event causing rollback. An event may perform two things that may have to be rolled back: it may modify the state of the logical process, and/or it may send event messages to other processes. Rolling back the state is accomplished by periodically saving the process's state, and restoring an old state vector on rollback. " Unsending" a previously sent message is accomplished by sending a negative or *anti-message* that annihilates the original when it reaches its destination. Messages corresponding to simulator events

are referred to as *positive* messages. In Time Warp, the smallest timestamp among all unprocessed event messages (both positive and negative) is called global virtual time (GVT). Irrevocable operations, such as I/O cannot be committed until GVT sweeps past the simulated time at which the operation occurred. The process of reclaiming memory and committing irrevocable operations is referred to as *fossil collection*.

There have been various enhancements to the standard optimistic approach such as lazy cancellation, lazy reevaluation, optimistic time windows, wolf calls, etc. All these enhancements can be used in conjecture with the prediction approach since most of the enhancements deal with handling causality errors. Both lazy cancellation [11], and lazy reevaluation [17] are mechanisms used to rollback when a causality error has been detected. The Wolf mechanism [16] and other direct cancellation methods [1, 9, 10] are not affected by the predictors. Lazy cancellation takes advantage of lookahead, even though the application was not explicitly programmed to exploit it. Unlike conservative approaches that require lookahead to be specified explicitly, lazy cancellation exploits lookahead in a way that is transparent to the application program. The disadvantage in exploiting lookahead in this manner is that the overhead is greater. Thus, even though optimistic methods do not set up for exploiting lookahead directly, some programming techniques which are application dependent are sometimes included to take advantage of lookahead [3].

The cost of roll-back is of primary concern while deciding how much better these parallel implementations are as when compared to their sequential counterparts. Thus, a critical assumption used by models is the cost of rolling back a logical process. Radically different results ensue depending on what is assumed.

4.2 Proposed Technique

The proposed technique : simulation of events based on predictions, has performance ranging from the Chandy-Misra approach to the Time Warp approach and was developed as part of a parallel simulation of a spacecraft's control commands [12]. The strategy can be viewed as an enhancement of the Time Warp strategy [15] on one hand and as an enhancement of the Chandy-Misra strategy on the other.

The idea is similar to lookahead technique for conservative methods but goes one step further. It provides a mechanism to partially avoid the causality errors before the simulation of an event starts. While the optimistic techniques *blindly* start the simulation of an event $Event_i$, and then check if any causality errors have occurred, the proposed technique predicts the global state variables that are used by $Event_i$ and then proceeds with the simulation. This decreases the probability of causality errors.

The predicted values for the state variables are stored and when the correct values of the state variables are received, the predicted values are compared with the correct values : in case of discrepancy, $Event_i$ is rolled back, and the event is reexecuted with the correct values of the state variables as is the case with all optimistic techniques. It is not necessary for the predictors to predict the values of all state variables - only the state variables required by $Event_i$ are predicted.

The proposed technique can also be considered as an enhancement of the conservative Chandy-Misra approach since we do not allow predictions to propagate to other events, as a result of which, rollbacks are restricted to single events only. The propagation of predicted values to other events might cause cascaded rollbacks, which would make the approach comparable to the Time-Warp approach. At present, we compare the optimistic Time-Warp approach with the proposed technique, considering only single event rollbacks in either case. For comparison purposes, we consider the prediction approach with 0% prediction to be an equivalent model of the Time-Warp approach. Thus, a brief look into the Time

Warp mechanism and the rollback costs associated with it seems befitting at this stage as a comparison between the performance of the proposed technique and the existing Time-Warp technique is desirable.

Rollback Costs in Time Warp:

By rollback cost we mean any computation that is not present during the normal, forward progress of the computation [8]. We will assume that the state of the process is saved at the end of the execution of each event, and state saving is achieved by copying the entire state vector for the process. This strategy is used in both Fujimoto's and JPL's current implementations of Time Warp, and *minimizes* the cost of rollback. Rollback entails three overheads:

1. Restoration of the input queue;
2. Restoration of the state queue;
3. Restoration of the output queue.

Restoration of the input and state queues incur negligible cost because only a few machine instructions are required to reset the pointer to the next event to be processed. The principal overhead in a rollback is to restore the output queue, which involves sending anti-messages, assuming aggressive cancellation is used. Since the principal overhead is the time required to send antimessages, it is reasonable to assume that the rollback cost is proportional to the length of the rollback. However, rolling back T time units will not take more time than forward progress by this amount because sending anti-messages takes less time than sending positive messages. Rollback requires only sending preexisting anti-messages; on the other hand, sending a positive message requires allocation of a message buffer and filling it with the data to be sent.

In the proposed technique, there are no antimessages since the correct values are compared with the predicted values : in case of discrepancy, the event is rolled back. This reduces the principal overhead in the time required for the rollbacks. The model specification developed at JPL has not been used in this simulation as the simulator at present only contains dummy events. In the case of sequence checking using models, rollback will be proportional to the number of models and not to the time. In this simulation though, we assume rollback to be be about 5% - 10% of the forward progress time for an event. For simulation purposes, we have assumed the rollback times for both Time-Warp and Prediction mechanisms to be the same.

Further, forward computation requires a number of other costs: scheduling overhead, processing incoming positive and negative messages, state saving and of course, the simulation computation itself. Thus, in practice, the rollback costs will be much smaller than that of the forward computation. The worst case is when there are relatively fine-grained event computations with negligible state saving overhead. We estimate the ratio of the time required for forward progress to that required for rollback to be approximately an order of magnitude for the implementation described in [6] - rollback is about one-tenth as costly as forward progress by an equivalent amount of simulated time. For larger-grained computations with significant state vectors, this ratio could easily be several orders of magnitude.

Thus, *the most appropriate cost for rollback using the assumptions described above is that it is proportional to the rollback distance but with a constant of proportionality less than 0.1* [6] (for medium-grained to large-grained events).

Prediction Mechanism:

The *event-list* is a set of event messages, with their associated times of transmissions, that are scheduled for the future. Every event in the event list has a set of state variables that might be dependent on state variables of other events.

The events start executing their code as and when possible without waiting for the correct values of the input variables to be passed down from events with smaller timestamps. However, the events do not start execution with the local values of the state variables; instead they *predict* the values of the input state variables based upon knowledge of previous events in the event list, their corresponding parameters, and the local values of the variables.

In the case of the prediction technique it is evident that the probability of rollback decreases with increase in the time spent in predicting the state variables of the event.

4.3 The Algorithm

The parallel machine used to simulate this algorithm is a 64 node Ncube. The machine does not have shared memory, therefore, the different nodes need to communicate with each other by means of message passing. The *forward-progress* time of $Event_i$ is the amount of time required to execute the event, provided the values of the state variables. The *prediction-time* of the event is the amount of time spent in predicting the values of the state variables of the event. The *rollback-time* is the amount of time spent in rolling back the event which was incorrectly executed. The probability-of-error factor provides the probability that predictions for an event are incorrect.

The assumptions made while implementing the prediction approach are the following:

- There is no shared memory, so processes communicate only via message passing.
- Each event has a timestamp that indicates its order in the global list of events.
- Each processor maintains a (consistent) global-clock which indicates the time upto which the simulation has been performed correctly in the global-list.
- All the processors involved in the simulation have a copy of the global-list of events with their timestamps.
- All events have equal grain size.
- A standard block of events consists of 100 events. The events in the actual event-list are executed block by block in the sense that once events 1 to 100 are executed correctly, the next 100 events are loaded into the Ncube processors and the same methodology of simulation is used. At present, we consider only one level of simulation. Since the number of rollbacks are reduced in the first level of simulation, the total number of rollbacks due to cascading is also reduced. Thus cascaded rollbacks will make a big difference in the performance of this approach. In future simulations, we do need to consider cascaded rollbacks in order to make the comparison complete.
- There is no load sharing involved, all events are randomly (but evenly) distributed at all processors of the Ncube.
- All events at every processor are sorted in increasing order of timestamps.

- Events do not generate new events that are added to the event-list i.e. we do not consider delay commands at present.

- The predicted results are not broadcasted - none of the other events use the predicted results as their state variables, which prevents the forward-propagation of errors in case of incorrect predictions.

At present, the simulator consists of simple skeleton code, with dummy events. The forward-progress, prediction-time, etc. have been implemented simply as empty loops. The counter of the loops can be changed to provide flexibility in the forward-progress, prediction and rollback times. Also, a random number generator provided with a certain probability-of-error is used to generate numbers between 1 and 100, and this is used to determine whether a prediction was correct or incorrect. The simulator has been tested with different values of forward-progress time, prediction-time, rollback-time, probability-of-errors, and also for different number of processors involved in the simulation. All different combinations have been used to evaluate the performance of the simulator. The pseudo-code of the algorithm is presented in Figure 1.

The details of the simulation are presented below. Each processor P_i allocated on the Ncube, initially gets the block of commands to be simulated and identifies the commands that P_i is responsible for simulating. The events in the event-block are distributed equally and randomly across all the processors. The events that P_i is responsible for simulating are stored in increasing time-stamp order in a local event-list at P_i. Now, every P_i starts processing events in its local event-list.

The events in the local event-list of P_i are processed as follows. P_i initializes the probability-of-error of correct execution of events, and the number of state variables to be broadcast after every correct execution. Then, P_i predicts the values of the state variables for the first event on its local list. Initially, for prediction, all processors assign the input values to their state variables where applicable or use the predictor to obtain some values. P_i then processes the event based on the predictions and stores the results obtained using predicted state-variables. Once the execution of the event is over, P_i checks to see if the first event on its local list is the first event in the global list too. If it is the current global-command, P_i checks to see if it had predicted its state variables correctly. If the predictions were correct, P_i predicts and processes the next command on its local list and then broadcasts the correct results of the current global-command. (Note that the processor which has the first event, definitely processes the event correctly since it is already provided with the inputs, and broadcasts the correct-results to all the processors.) If the predictions of P_i were incorrect, then P_i reexecutes the current command with the correct values of the state variables and then broadcasts the results. After the broadcast in either case, P_i advances the global-clock indicating that events upto and including the current global-command have been processed correctly. The pointer in the global-list is advanced to point to the next command in the global-list.

While P_i is broadcasting the results, the other processors check their buffer to verify if there is any information to be read. If there is information to be read, the processors update the values of the state variables with the correct values that they have received, update their global clock and proceed to execute the next command on their local lists. In the case where there is no information to be read, they proceed to execute the next command n their local event-lists. This basic procedure is repeated until the global-clock has the value of the last event in the event-block indicating that all the events in the event-block have been processed correctly.

Start of Main Loop

Each processor P_i

[

Gets block of commands to be simulated (global-list)

Identifies commands P_i is responsible for simulating

Saves the events in increasing timestamp order in a local event-list

Process-events in local event-list

]

End of Main Loop

The last step of the code above is enumerated below to provide insight into how each processor processes events in its local list.

Start of Process-Events Loop

Initialize probability-of-error, and # of state variables to be broadcasted after every correct execution

Prediction(first event)

Forward-Progress(first event)

Store-Predicted-Results(first event)

While(# of events processed correctly < total # of commands in event-block) do

[

Processor checks to see if it has executed the current command in the global-list $Event_j$

If(processor has executed the current command in the global-list) do

[

If(processor has predicted its state variables correctly for $Event_j$) do

[

Prediction(next event)

Forward-Progress(next event)

Store-Predicted-Results(next event)

]

else do

[

Rollback($Event_j$)

Forward-Progress($Event_j$)

]

Processor upgrades global-clock = timestamp of $Event_j$

Broadcast correct state variables + Global clock

]

else do

[

If(Processor has message to read in message buffer) do

[

Processor saves the correct state variables, and upgrades its global-clock

]

Prediction(next event)

Forward-Progress(next event)

Store-Predicted-Results(next event)

]

]

End of Process-Events Loop

Figure 1: Pseudo-Code of the Algorithm

5 Model Development

The following is a performance model for the simulation running on the hypercube architecture. The model is based on work by Roger D. Chamberlain and Mark A. Franklin [6] that considers hierarchical discrete event simulation on a hypercube architecture. The model variables are defined below.

- R_P : simulation runtime using P processors;
- E : number of events to be simulated;
- α : work distribution across communication links;
- P : number of processors;
- t_E : single event processing time;
- t_{CF} : CPU time for single message formulation;
- t_{CT} : CPU time for single message transmission;
- t_{CR} : CPU time for single message reception;
- t_{LM} : Link time for message transmission;
- t_{LV} : Link time for single message protocol overhead;
- H : average number of hops required per message;
- t_{CPU} : total CPU time;
- t_{COMM} : total communication time;
- t_{SYNC} : total synchronization time;
- t_{PRED} : time for prediction of event;
- t_{READ} : time for reading in correct values;
- t_{BROAD} : time for broadcasting correct values to all processors;
- t_{ROLL} : time for rolling back an event;
- P_{err} : the probability of processing an event incorrectly;
- T_{pred} : Percentage of forward progress time that prediction requires;
- T_{read} : Percentage of forward progress time that reading in the correct values requires; and
- T_{roll} : Percentage of forward progress that rollback requires.

The simulation time for E events on P processors, R_P is defined as

$$R_P = MAX(t_{CPU}, t_{COMM}) + t_{SYNC}$$

where t_{CPU}, t_{COMM}, and t_{SYNC} are as defined below.
t_{CPU} = event-evaluation time + communications overhead for CPU

$$t_{CPU} = \frac{E}{P}[t_{PRED} + t_E + t_{READ}] + P_{err} \cdot E[t_{ROLL} + t_E] + E \cdot t_{BROAD}$$

Note that once a processor realizes that it has executed an event incorrectly (it does so on receiving the correct values of the predicted variables), it reexecutes the event using the correct values of the variables. Each event that has incorrect predicted variables need to be rolledback and reexecuted again resulting in the $P_{err} \cdot E[t_{ROLL} + t_E]$ term in the equation for t_{CPU}.

Assuming t_{CF}, t_{CT}, and t_{CR} as the CPU time to formulate, transmit and receive a message, t_{BROAD} gives the time spent by each processor for communication. Note that this equation introduces a communication imbalance factor α to account for unequal distribution of message volume across processors.

$$t_{BROAD} = \alpha \cdot \frac{E}{P}[H(t_{CT} + t_{CR}) + t_{CF}]$$

The first term in the above equation corresponds to the work that must be performed on all messages, either original or pass-through. The second term accounts for the work that must be performed when a processor originates a message.

We now consider t_{COMM}. The total communication time is given by

$$t_{COMM} = \alpha \cdot H \cdot E[\frac{t_{LM} + t_{LV}}{W}]$$

The average communications width W in the above equation is the number of simultaneous messages that can be transmitted concurrently. The communications width is bounded by the number of bidirectional links at each processor.

The third quantity to consider is the synchronization time t_{SYNC}. If the synchronization is performed via a complete exchange, each processor must send and receive a message in each cube dimension. The expression for t_{SYNC} can be developed assuming that log P message times are required. Thus,

$$t_{SYNC} = t_{LV} log P + t_{LM}(P - 1).$$

6 Analysis of Results : Prediction vs Time-Warp

The performance of the prediction mechanism has been compared with the performance of the Time-Warp mechanism and the results are summarized below. We assume that in both methodologies, the rollbacks are not cascaded. In other words, we assume that the error in an event is not propagated to other events as well. In the prediction technique, we assume that no processor starts executing events with values received from other events which executed based upon predicted values. Thus, the effect of predictions is restricted to the event itself and not propagated to other events.

By introducing the concept of prediction, the main objective is to reduce the probability of error of the correct execution of events. Thus, the more the amount of time spent in prediction, the lesser the probability of error. Moreover, the performance of this technique also depends upon the particular application which uses the method. For example, if we consider sequence checking of spacecraft control commands, this method would fare very well since the predictions would be almost always correct. Depending on statistics gained over a period of time, the values of the variables can be predicted with considerable accuracy.

The results gained from simulation run on the skeleton code are summarized below. The simulations with zero prediction correspond to the optimistic Time-Warp approach (with no cascaded rollbacks implemented at present). Thus, a comparison is made between the Time-Warp method (prediction = 0) and the Prediction method (with prediction = 5%, 10%, and 15%).

SPEEDUP with rollback and prediction taking 5% and 0% of forward progress, respectively				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
1	1.93	3.70	6.84	11.40
5	1.81	3.63	6.56	11.27
10	1.68	3.31	6.27	9.73
20	1.58	3.13	5.67	9.20
30	1.33	2.65	4.78	8.89

SPEEDUP with rollback and prediction taking 5% and 5% of forward progress, respectively				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
1	1.84	3.54	6.63	10.93
5	1.73	3.47	6.25	10.40
10	1.61	3.18	6.02	9.44
20	1.53	2.86	5.65	9.01
30	1.29	2.54	4.73	8.75

SPEEDUP with rollback and prediction taking 5% and 10% of forward progress, respectively				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
1	1.76	3.37	6.36	10.63
5	1.66	3.35	6.03	10.24
10	1.55	3.05	5.78	9.27
20	1.48	2.76	5.44	8.73
30	1.25	2.48	4.57	8.64

The results can be analyzed as follows. Consider the speedup obtained with rollback and prediction

taking 5% and 0% of forward-progress respectively. This would correspond to the speedups obtainable using Time-Warp since no time is spent in prediction. Now, consider the speedups obtained with rollback and prediction requiring 5% and 5% of forward progress respectively. Since some amount of time is spent in prediction, the probability-of-error of execution of events should decrease. For example, assume that due to prediction the probability-of-error drops from 20% to 10%. The speedups obtained with prediction (using 2, 4, 8, and 16 processors) are 1.73, 3.38, 6.16 and 9.91 as compared to the Time-Warp values of 1.59, 2.97, 5.85, 9.37. In the case where 10% of forward progress is spent in prediction assume that the probability-of-error falls to 1% from 20% for Time-Warp. Then, the speedups using prediction are 1.76, 3.39, 6.45, and 10.67 respectively. From the analysis it is evident that the prediction technique performs better than the Time-Warp method, under the above assumptions.

For the initial simulations, whose results have been presented in this paper, we assume that the predicted values are not propagated to other events. Thus rollback involves single events only. In future, when we consider cascaded rollbacks for the comparison of the methods, it is possible that the prediction approach will perform better as the number of events which will have to be rolled back will be lesser than in the case of Time-Warp. In the case of single event rollbacks, we assume that by spending time in prediction, we can reduce the probability of error P_{err-TW} of the Time-Warp method considerably to $P_{err-pred}$ for the Prediction method. Thus, when cascaded rollbacks are considered, we will only have to rollback the events (and other related events) that correspond to $P_{err-pred}$ rather than the events which correspond to P_{err-TW} (which is greater than $P_{err-pred}$).

The results in the remaining tables presented below, for rollback times of 10% and 15% can also can be analyzed similar to the above manner and no further analysis is presented here.

SPEEDUP with rollback and prediction taking 10% and 0% of forward progress, respectively				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
1	1.92	3.61	6.83	10.70
5	1.80	3.60	6.50	10.48
10	1.67	3.28	6.22	9.70
20	1.56	2.96	5.60	8.88
30	1.31	2.60	4.71	8.68

SPEEDUP with rollback and prediction taking 10% and 5% of forward progress, respectively				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
1	1.83	3.53	6.52	10.48
5	1.73	3.40	6.22	10.34
10	1.60	3.20	5.91	9.40
20	1.52	2.83	5.51	8.75
30	1.27	2.50	4.63	8.54

SPEEDUP with rollback and prediction taking 10% and 10% of forward progress, respectively				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
1	1.75	3.38	6.28	10.29
5	1.65	3.33	6.02	10.11
10	1.54	3.10	5.73	9.20
20	1.46	2.73	5.34	8.54
30	1.23	2.44	4.52	8.45

SPEEDUP with rollback and prediction taking 15% and 0% of forward progress, respectively				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
1	1.92	3.56	6.67	10.79
5	1.80	3.55	6.18	10.65
10	1.66	3.23	6.13	9.54
20	1.56	2.95	5.54	8.94
30	1.30	2.56	4.72	8.62

SPEEDUP with rollback and prediction taking 15% and 5% of forward progress, respectively				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
1	1.83	3.43	6.52	10.69
5	1.72	3.40	6.01	10.51
10	1.61	3.10	5.94	9.25
20	1.50	2.80	5.42	8.77
30	1.25	2.48	4.61	8.48

SPEEDUP with rollback and prediction taking 15% and 10% of forward progress, respectively				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
1	1.75	3.38	6.08	10.36
5	1.65	3.35	5.83	10.13
10	1.53	3.01	5.74	8.81
20	1.45	2.70	5.35	8.63
30	1.21	2.41	4.50	8.36

The graphs presented in Appendix B correspond to constant rollbacks and the algorithm being executed on a constant number of processors. The speedups obtained for varying probability of error and time of prediction are plotted in the graphs. By analyzing the graphs, the influence of prediction

on the error probabilities for constant speedups can be determined and is summarized below in the following tables. The tables contain information regarding how much the probability of error needs to be reduced by the prediction mechanism to obtain equivalent performance to the Time-Warp mechanism. The graphs presented in Appendix C correspond to constant probability of errors and varying rollback times. It can be observed from the graphs that as the time for rollback increases, the speedup decreases which is as anticipated.

Reduction in P(error) to obtain comparable performance to Time-Warp (Pred = 5% & Rollback = 5%)				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
5	3%	5%	3%	5%
10	3%	2%	5%	2%
20	7%	3%	2%	5%
30	2%	3%	2%	6%
50	1%	1%	1%	2%

Reduction in P(error) to obtain comparable performance to Time-Warp (Pred = 10% & Rollback = 5%)				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
5	5%	5%	5%	5%
10	5%	4%	8%	3%
20	12%	7%	7%	10%
30	3%	6%	4%	14%
50	1%	2%	1%	3%

Reduction in P(error) to obtain comparable performance to Time-Warp (Pred = 5% & Rollback = 10%)				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
5	3%	3%	4%	4%
10	3%	1%	5%	1%
20	6%	2%	3%	3%
30	2%	3%	1%	8%
50	1%	1%	1%	2%

Reduction in P(error) to obtain comparable performance to Time-Warp (Pred = 10% & Rollback = 10%)				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
5	5%	5%	5%	5%
10	6%	3%	7%	2%
20	12%	4%	7%	6%
30	4%	6%	3%	13%
50	1%	2%	1%	3%

Reduction in P(error) to obtain comparable performance to Time-Warp (Pred = 5% & Rollback = 15%)				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
5	3%	5%	1%	5%
10	2%	3%	6%	1%
20	4%	6%	2%	3%
30	2%	3%	2%	5%
50	1%	1%	1%	2%

Reduction in P(error) to obtain comparable performance to Time-Warp (Pred = 10% & Rollback = 15%)				
Prob. of error	# of processors			
	2 proc.	4 proc.	8 proc.	16 proc.
5	5%	5%	5%	5%
10	5%	4%	9%	2%
20	6%	8%	5%	6%
30	4%	5%	4%	10%
50	2%	2%	1%	3%

The above tables do not correspond to accurate reductions required for equivalent performance of the two approaches. Approximate estimations have been made from observing graphs. We anticipate, that the reduction reduced is going to be lesser as the probability of error increases. Most of the inconsistencies arise from the Ncube which does not perform consistently, due to the different allocations of processors for each simulation run and the number of users accessing the system. We propose to perform a larger number of simulation runs to arrive at accurate values for the above tables.

7 Discussion of Enhancements

The standard optimistic simulation approach can be viewed as the prediction approach with zero predictors thus reducing the prediction approach to the standard optimistic approach. In the case of the standard optimistic approach, the probability of causality error occurring is largest since the local values of the variables are used in the computations : no time is spent on predicting the values of the variables.

Conceptually, the more the time spent predicting the values of the state variables of the current event $Event_i$, the smaller is the probability of causality error. In the prediction technique it is obvious that a causality error occurs only when the prediction of a state variable is incorrect. Since the predictors are stored in the memory of each logical process for comparison purposes, the distributed environment is most suited for this technique.

We propose two different approaches for prediction of state variables. In the first method, for each of the state variables in an event $Event_i$, we identify the previous event $Event_j$ in the event-list (above $Event_i$) which might modify them. Once the event is identified, for the prediction of each of the state variables that are needed for the simulation of $Event_i$, we simply compute the values of the variables from the corresponding code in $Event_j$. Note that we are interested only in the final value of the variable when exiting $Event_j$; any number of internal modifications of the variable in $Event_j$ are not of concern to us at present, but are required when we consider cascaded rollbacks in future.

Some events may be put on the event list by the other simulated events. Since this is not known *a priori*, the predictors should be used with lookahead method. In the worst case, the predictor will start checking backwards the entire event list; before the predictor is through with checking all the events there is a possibility that the local processor will receive the actual values of the variables it is predicting. In such a scenario, we simply disregard the prediction and start simulating the event using the actual state variables received. This boundary case corresponds to the standard conservative method (Chandy-Misra [7]). The prediction can be stopped after some time and the current value is the predicted value or one can examine only the portion of an event that can affect the predicted variable. The best prediction method is application dependent and several simulations and analysis are on their way to determine the limits and expectation of these predictions.

Another method that can be used for prediction is based on statistical information obtained from previous runs of the simulator. The prediction for each event $Event_i$ in this method, is simply the most common value of the state variable at the beginning of the simulation of this event (data maintained from previous runs). One can go even one step deeper, and check if there is any correlation between the previous event (or the sequence of them) that update this variable and its value at the beginning of the simulation of event $Event_i$. Note that a combination of the first and second methods can be used for the prediction.

We have to be aware that after the execution of the event the values of the *output state variables* should be sent to the *next* event that uses them as *input state variables*. Also they must be marked as variables based upon predictions rather then the actual input variables. At some point of a time the logical processor will receive all the actual values of its predicted variables and check them against its predictions. If the predictions were correct then it sends a message to the next user of its output variables informing the event that its output variables were correct.

There are several *load balancing* strategies we propose to examine to minimize information exchange between nodes of hypercube. The proposed techniques range from the simple technique: the i^{th} node will pick up the (i *mod* n) commands where n is the number of nodes of hypercube; to more sophisticated technique that puts a reasonable number of dependent commands on the same node of hypercube such that this node will not execute longer than others.

8 Conclusions and Future Work

We have presented the concepts of the model generator, the sequential simulator, the existing conservative and optimistic methods of PDES briefly, and our proposed technique and algorithm in detail. The results of the simulations run on the testbed on the Ncube have been presented and we have performed a comparison between the Time-Warp approach and the prediction approach illustrating cases where the prediction approach performs superior to the Time-Warp approach and case where the vice versa is true. The further enhancements that can be made to the prediction approach to improve its performance have been discussed.

Future work which can be done to enhance the performance of the prediction mechanism has been briefly summarized below. For testing our parallel implementation we need to check several combinations of load balancing techniques with empty or complete predictors with synchronous or asynchronous implementations of message exchanges. This is the core of short time future work that will give sufficient knowledge to proceed to complete checker and future checkers for new spacecrafts.

In future implementations, several other load balancing techniques will be tested. Some proposed methods are application dependent that are based upon identifying what commands are *independent* (do not use the same variables as input/output) and assigning them to different processors. This will ultimately reduce the number of communication messages needed, since a large percentage of the *dependent* commands will be placed on the same processor on adjacent (sharing hardware links) processors. The latter is especially crucial for implementing this on distributed hardware that does not support broadcast operation directly.

Currently, the processors do not rollback its simulation until they get all the actual values of the output variables from the previous commands on the list. In future, the rollback after receiving only a portion of its actual incorrect input predicted variables (any where from 1 to all) for a command will be tested. There are no definite plans what to do when the output variables from the previous commands on the list based upon their predictions of their input variables are received.

Ultimately, the ability to write a good predictor will determine the efficiency of the parallel simulation.

An algorithm for semi-automatically generating models for the spacecraft has been presented. By using the algorithm, most of the models can be generated though not all of them. Once the individual models have been generated, the interdependencies among the models need to be determined. There are instances where one model might trigger other models either through a delay, or directly. Such models must be merged together by having transitions which are activated conditionally from one model to the other. Algorithms for efficient merging need to be developed.

9 Acknowledgments

The authors would like to thank T. Tang, L. P. Perry, Leon Alkalaj, R. C. Cole, A. V. Amador and C. Schalk for comments and discussions. The authors are especially thankful to C. Schalk, T. Tang and L. P. Perry for making the first prototype a reality. The authors are indebted to Joan Horvath for her invaluable help, numerous discussions and comments. This work was performed at Texas A&M under the auspices of the JPL Director's Discretionary Fund, under contract RFP#LO1-4125 with the National Aeronautics and Space Administration. The second author is indebted to NASA/ASEE Summer Faculty Fellowship which supported this project.

References

[1] M. Abrams. The object library for parallel simulation (olps). In *Winter Simulation Conference*, pages 210–219, December 1988.

[2] Leon Alkalaj. Towards a specification language and programming environment for concurrent constraint validation of spacecraft commands. JPL Technical Report, July 1992.

[3] D. Baezner, J. Cleary, G. Lomow, and B. Unger. Algorithmic optimizations of simulations on time warp. In *SCS Multiconference on Distributed Simulation 21*, volume 2, pages 73–78, March 1989.

[4] O. Berry. *Performance evaluation of the Time Warp distributed simulation mechanism.* PhD thesis, University of Southern California, May 1986.

[5] R.E. Bryant. Simulation of packet communications architecture computer systems. Technical Report MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.

[6] Roger D. Chamberlain and Mark A. Franklin. Hierarchical discrete-event simulation on hypercube architectures. In *IEEE MICRO*, pages 10–19, August 1990.

[7] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Eng. SE-5*, 5:440–452, September 1979.

[8] R. M. Fujimoto. Parallel discrete event simulation. *Communication of ACM*, 33(10):31–53, 1990.

[9] R.M. Fujimoto. Time warp on a shared memory multiprocessor. *Trans. Soc. for Comput. Simul. 6*, 3:211–239, July 1989.

[10] R.M. Fujimoto. Performance of time warp under synthetic workloads. In *SCS Multiconference on Distributed Simulation 22*, volume 1, pages 23–28, January 1990.

[11] A. Gafni. Rollback mechanisms for optimistic distributed simulation systems. In *SCS Multiconference on Distributed Simulation 19*, volume 3, pages 61–67, July 1988.

[12] J. C. Horvath, A. Kanevsky, T. Tang, L. P. Perry, R. C. Cole, A. V. Amador, and C. Schalk. Checking critical constraints on the hypercube. Submitted to publication, November 1990.

[13] J.C. Horvath, Leon Alkalaj, A. Kanevsky, and A. Amador. Hypercubes for critical spacecraft command verification. In *JPL Annual Report*, pages 377–381, June 1992.

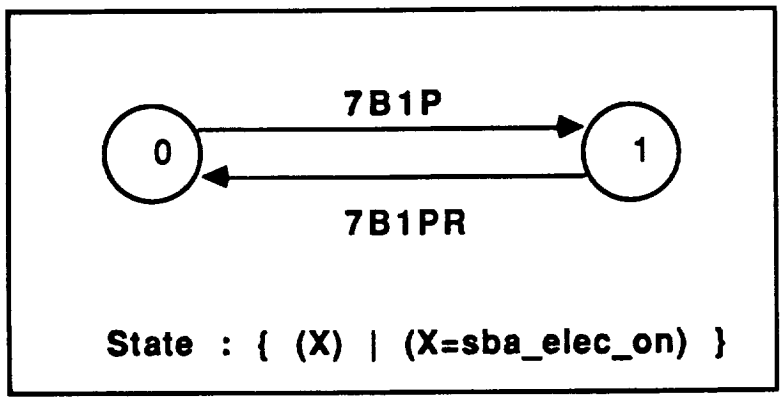
[14] D.R. Jefferson and H. Sowizral. Fast concurrent simulation using the time warp mechanism, part i: Local control. Technical Report N-1906-AF, RAND Corportation, December 1982.

[15] A. Kanevsky. Sequence checking in parallel: Issues, problems, solutions. JPL Seminar, August 1990.

[16] V. Madisetti, J. Walrand, and D. Messerschmitt. Wolf: A rollback algorithm for optimistic distributed simulation systems. In *1988 Winter Simulation Conference*, pages 296–305, December 1988.

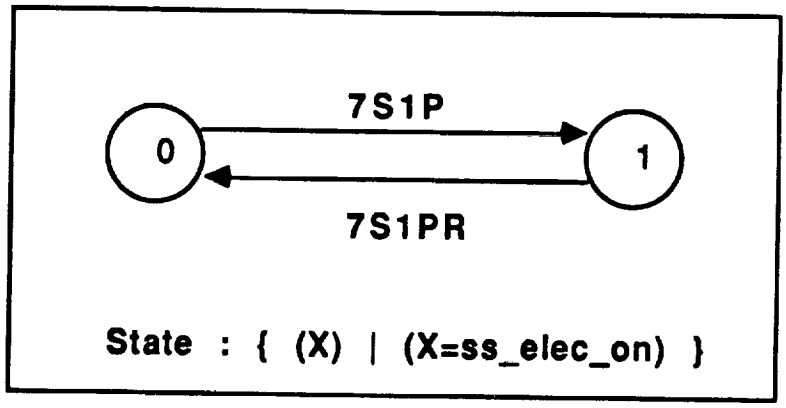
[17] D. West. Optimizing time warp: Lazy rollback and lazy re-evaluation. Master's thesis, University of Calgary, January 1988.

Appendix A



	SBA Electronics Actions	
	7B1P (on)	7B1PR(off)
S0		error
S1	error	

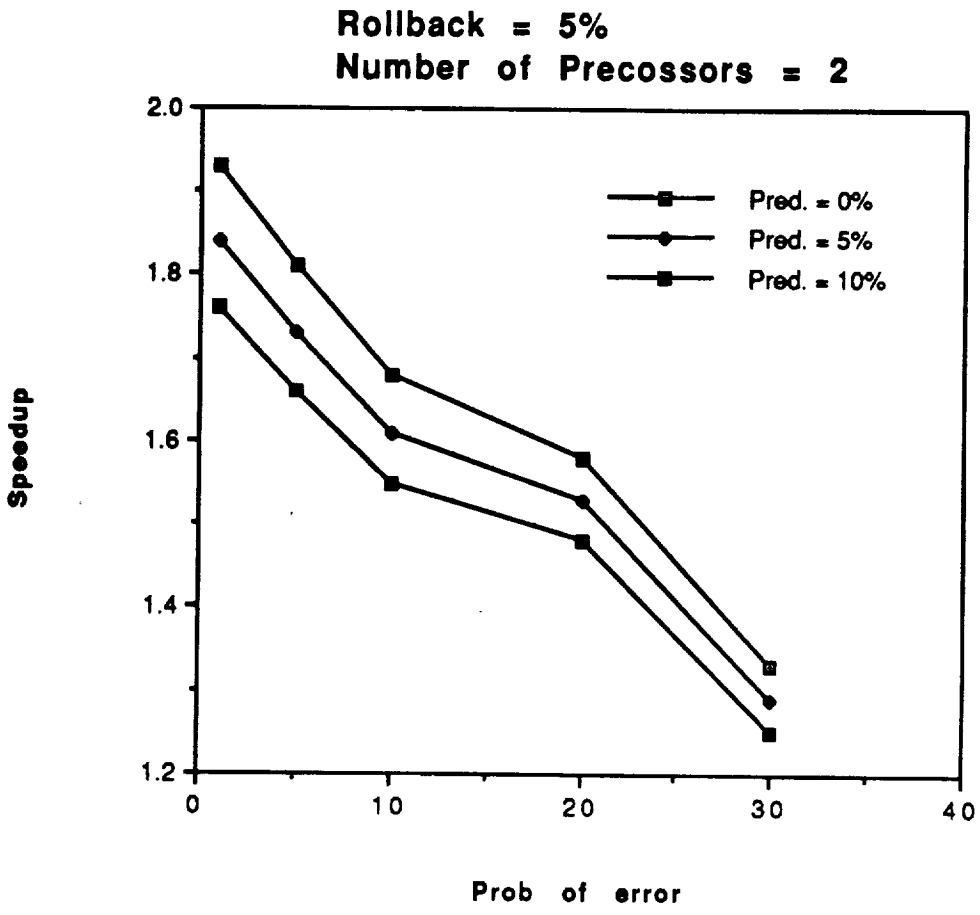
SBA Electronics Specification : State Model and Action Table

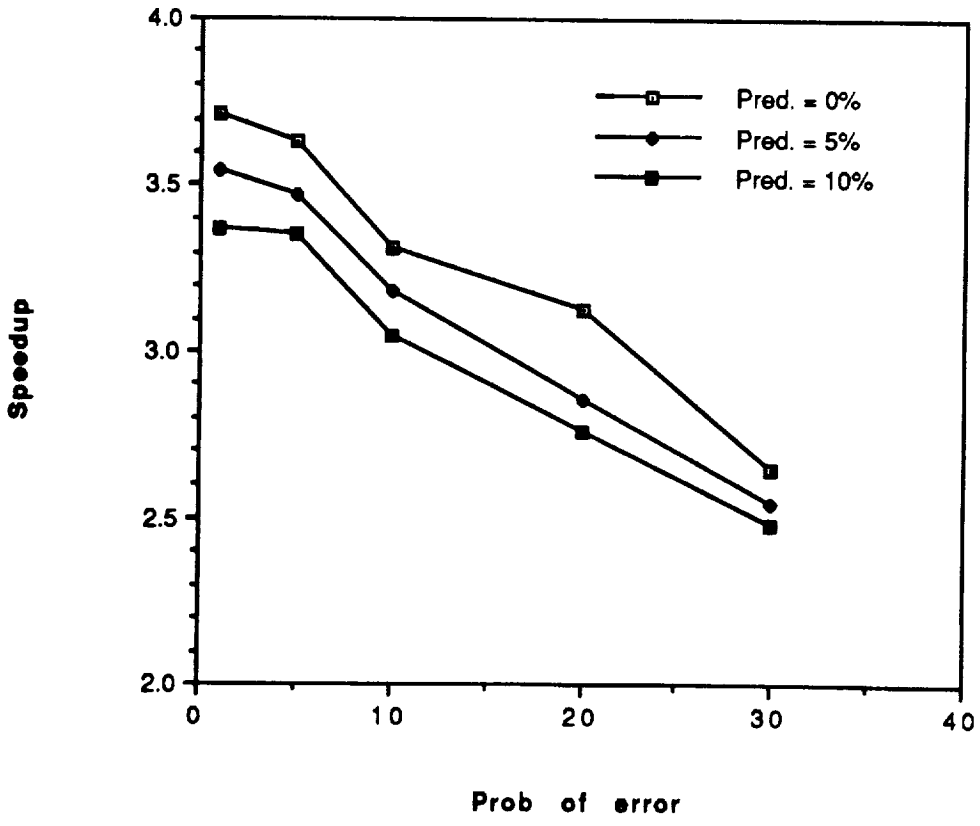


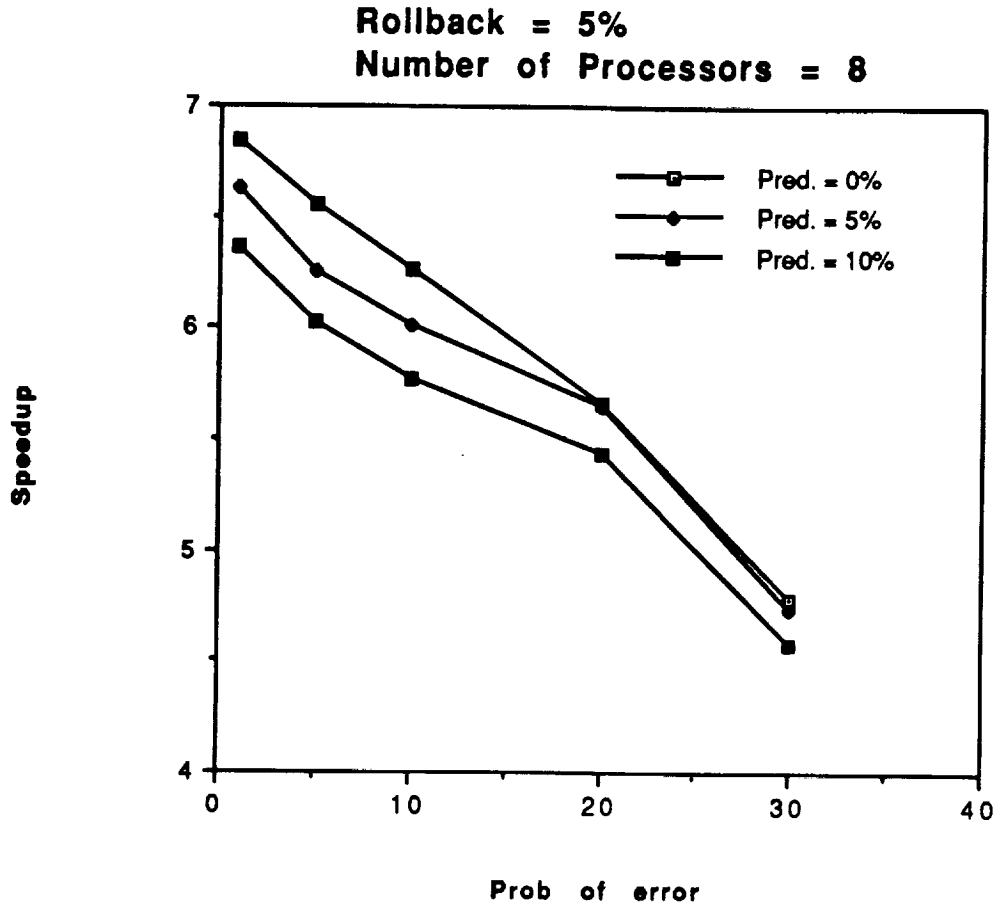
Star Scanner Electronics Actions		
	7S1P (on)	7S1PR(off)
S0		error
S1	error	

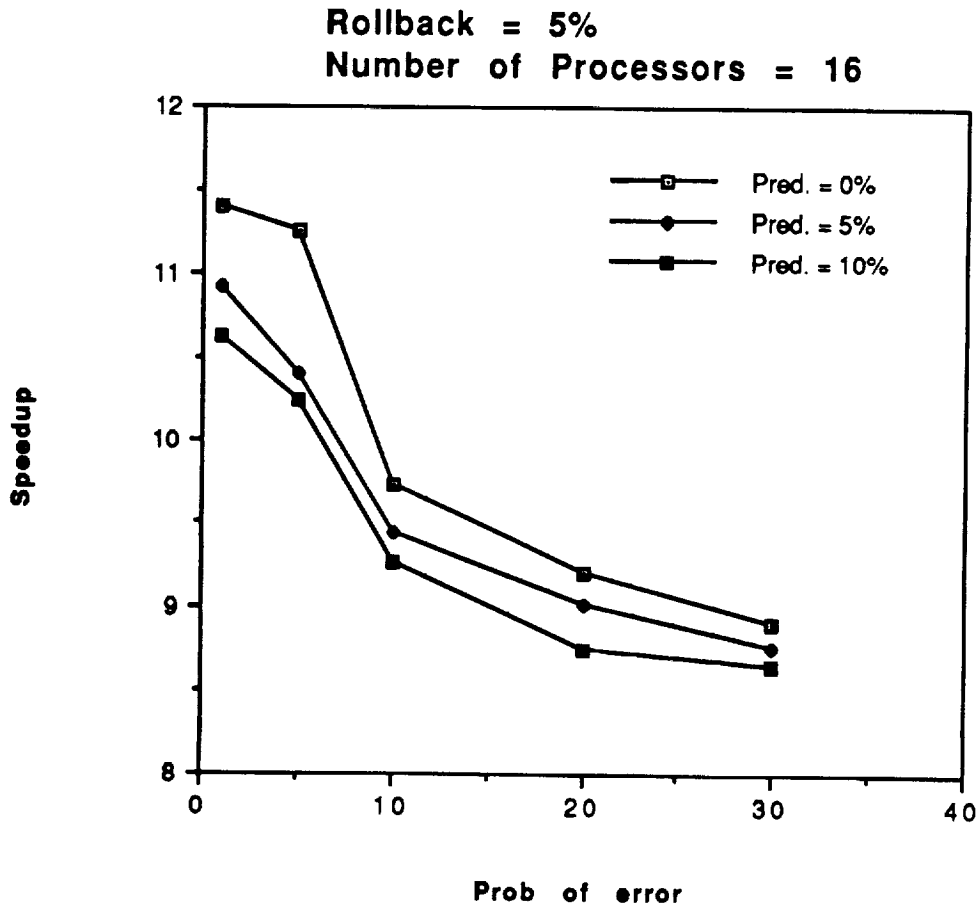
Star Scanner Electronics Specification : State Model and Action Table

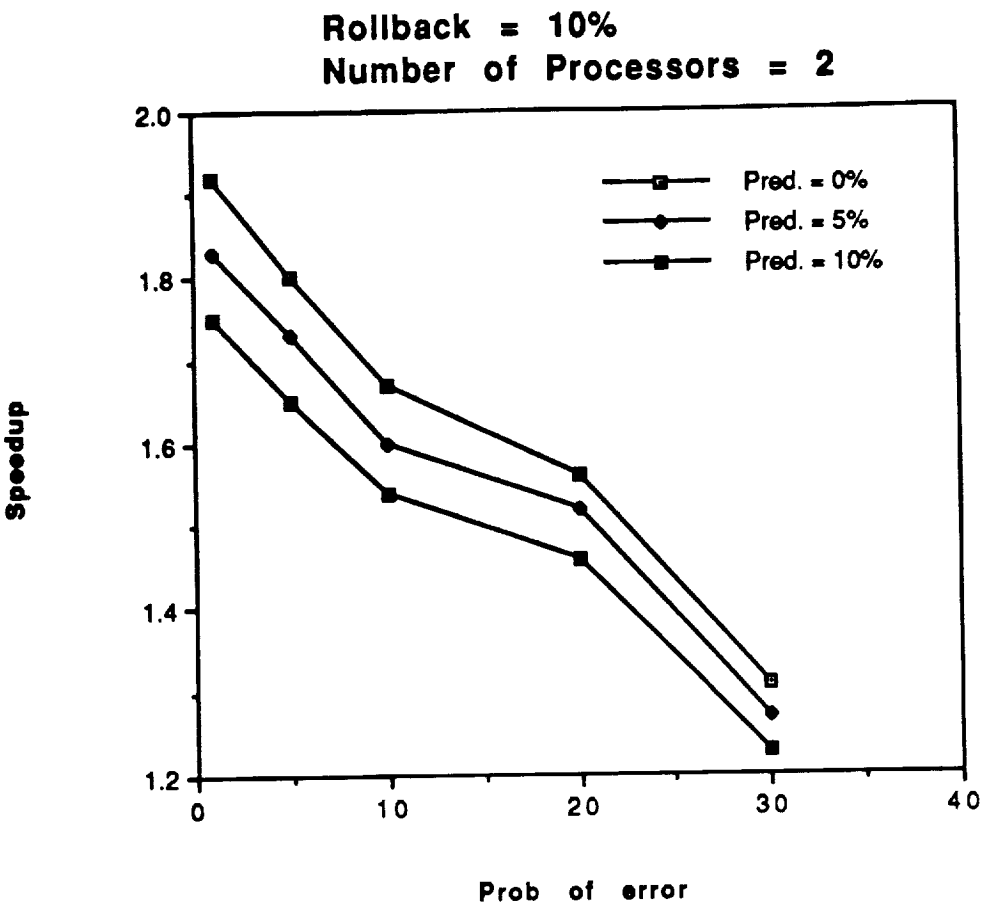
Appendix B

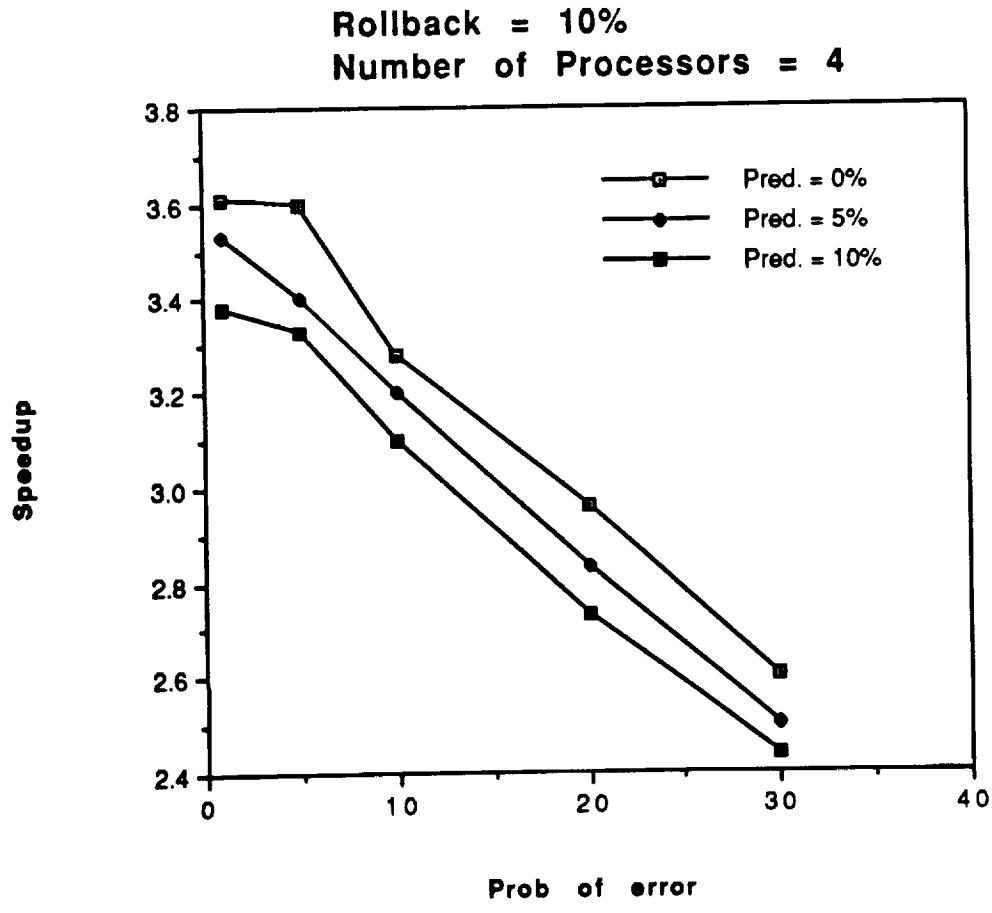

Rollback = 5%
Number of Precossors = 2
Speedup
2.0
1.8
1.6
1.4
1.2
0
10
20
30
40
Pred. = 0%
Pred. = 5%
Pred. = 10%
Prob of error

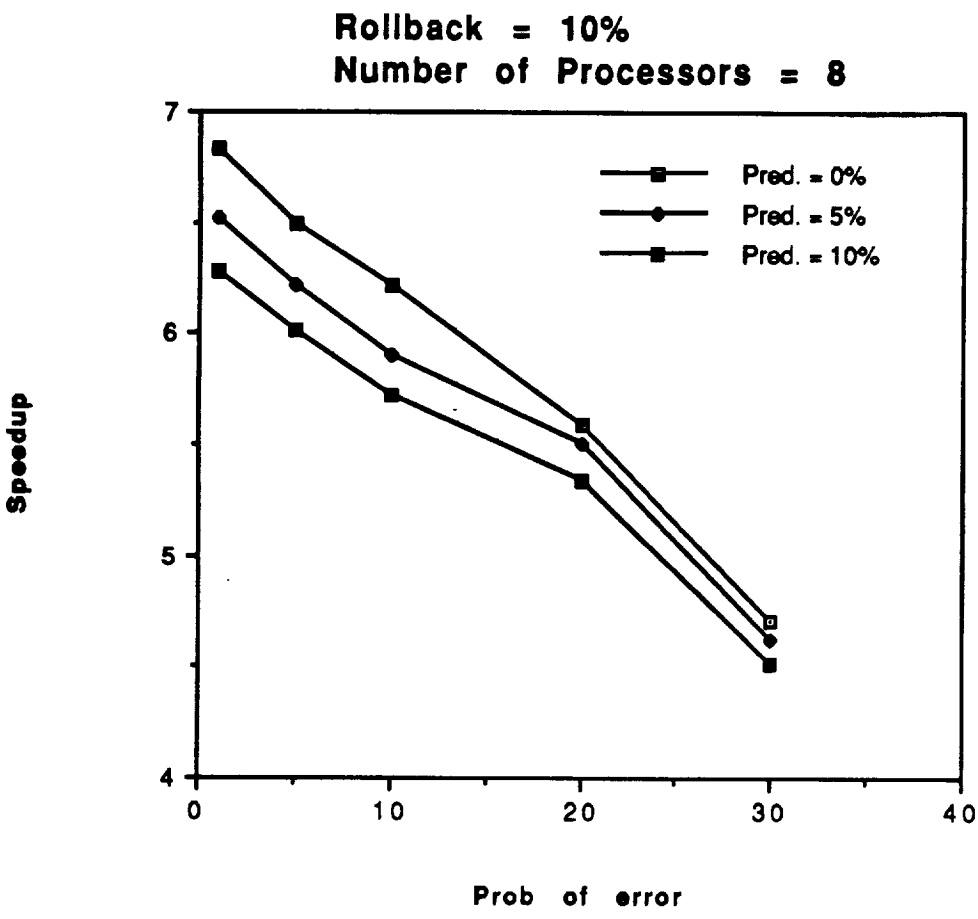

Rollback = 5%
Number of processors = 4
Speedup
4.0
3.5
3.0
2.5
2.0
0
10
20
30
40
Pred. = 0%
Pred. = 5%
Pred. = 10%
Prob of error

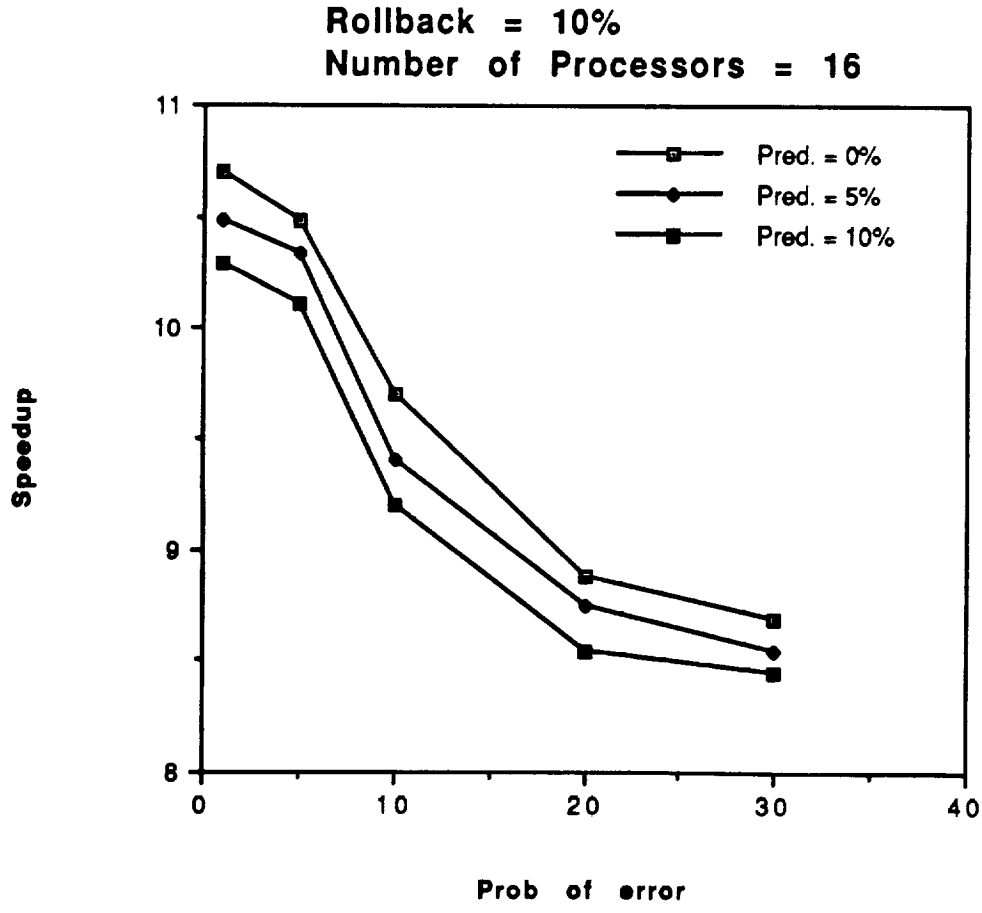

Rollback = 5%
Number of Processors = 8
Pred. = 0%
Pred. = 5%
Pred. = 10%
Speedup
7
6
5
4
0
10
20
30
40
Prob of error

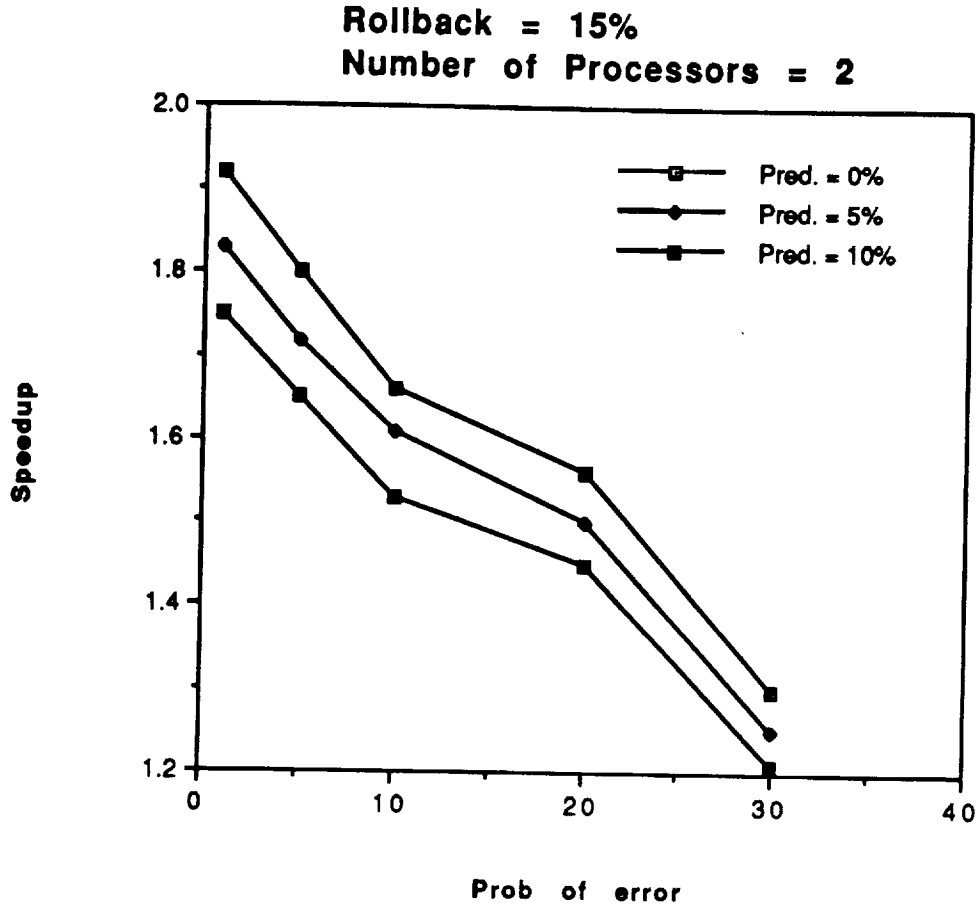

Rollback = 5%
Number of Processors = 16
Pred. = 0%
Pred. = 5%
Pred. = 10%
Speedup
12
11
10
9
8
0
10
20
30
40
Prob of error

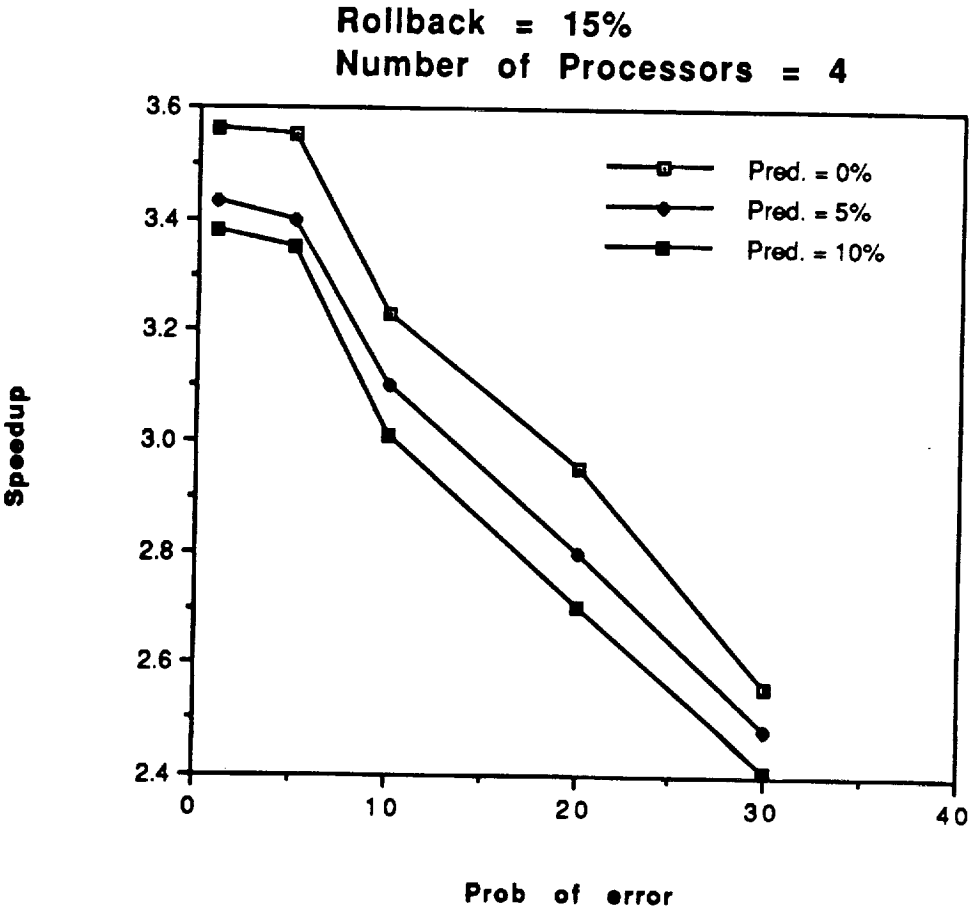

Rollback = 10%
Number of Processors = 2
Pred. = 0%
Pred. = 5%
Pred. = 10%
Speedup
2.0
1.8
1.6
1.4
1.2
0
10
20
30
40
Prob of error

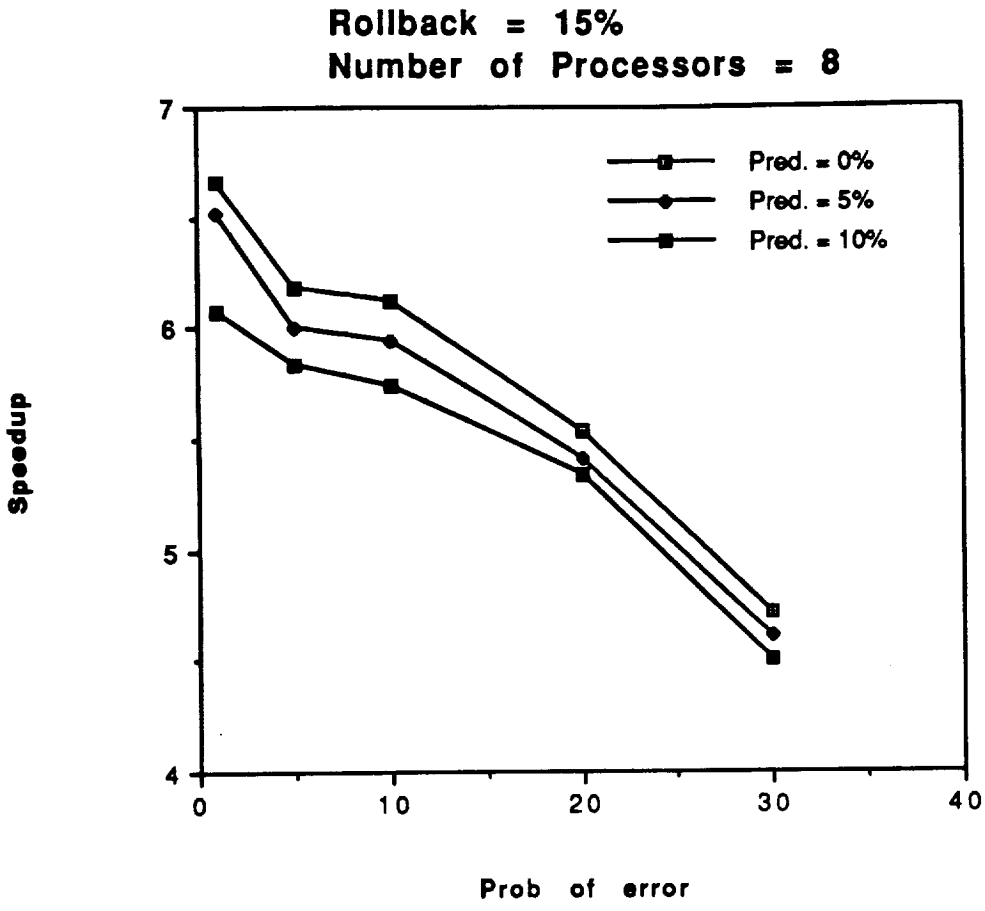

Rollback = 10%
Number of Processors = 4
Pred. = 0%
Pred. = 5%
Pred. = 10%
Speedup
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
0
10
20
30
40
Prob of error

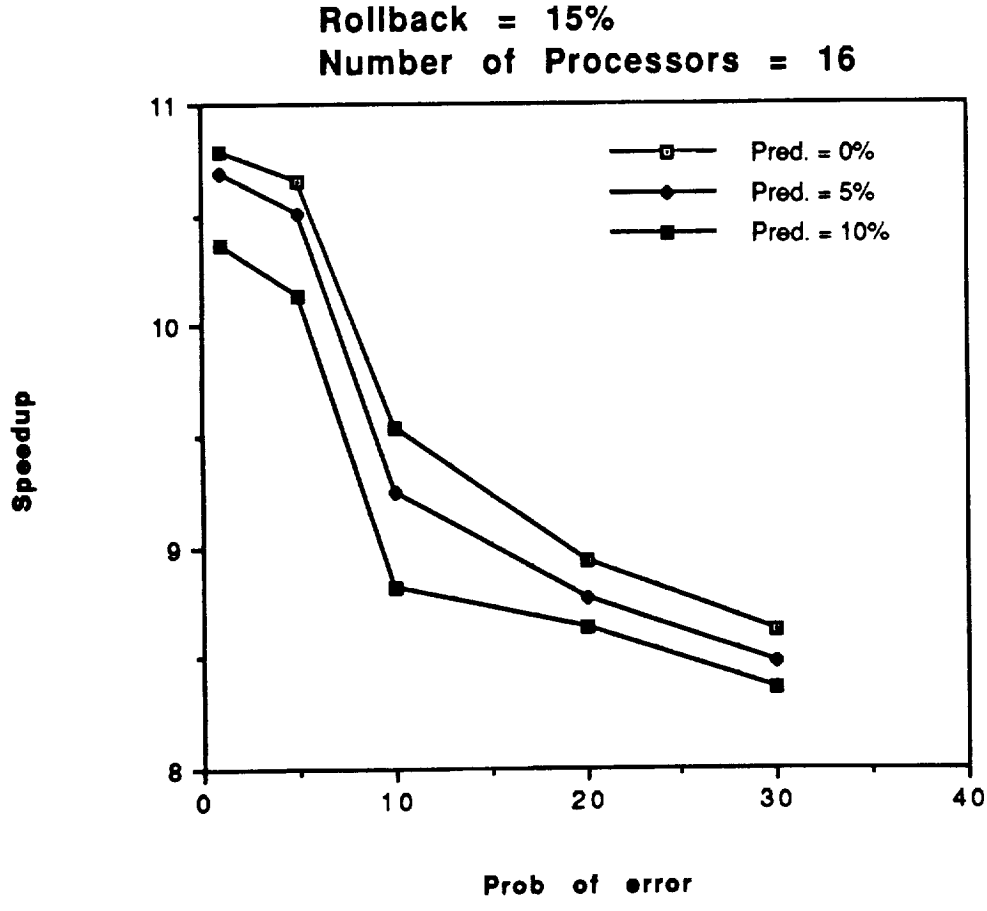

Rollback = 10%
Number of Processors = 8
Pred. = 0%
Pred. = 5%
Pred. = 10%
Speedup
Prob of error
7
6
5
4
0
10
20
30
40


Rollback = 10%
Number of Processors = 16
Pred. = 0%
Pred. = 5%
Pred. = 10%
Speedup
Prob of error
11
10
9
8
0
10
20
30
40

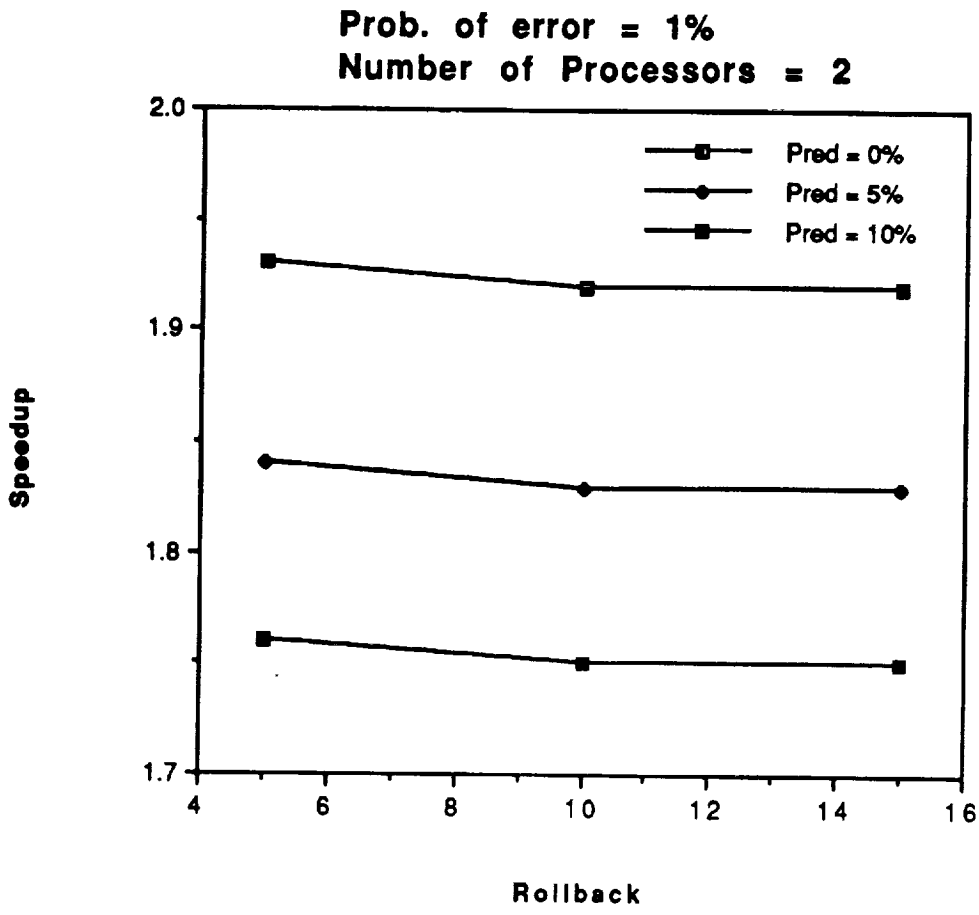

Rollback = 15%
Number of Processors = 2
Pred. = 0%
Pred. = 5%
Pred. = 10%
Speedup
2.0
1.8
1.6
1.4
1.2
0
10
20
30
40
Prob of error

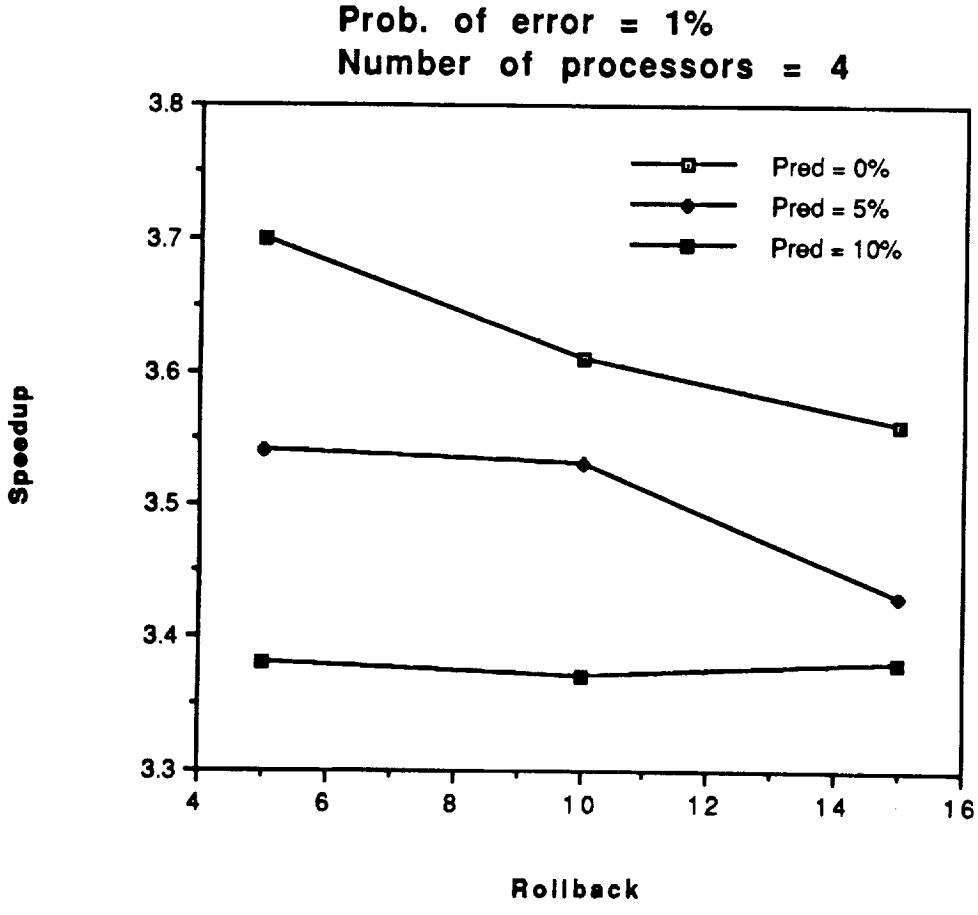

Rollback = 15%
Number of Processors = 4
Pred. = 0%
Pred. = 5%
Pred. = 10%
Speedup
3.6
3.4
3.2
3.0
2.8
2.6
2.4
0
10
20
30
40
Prob of error

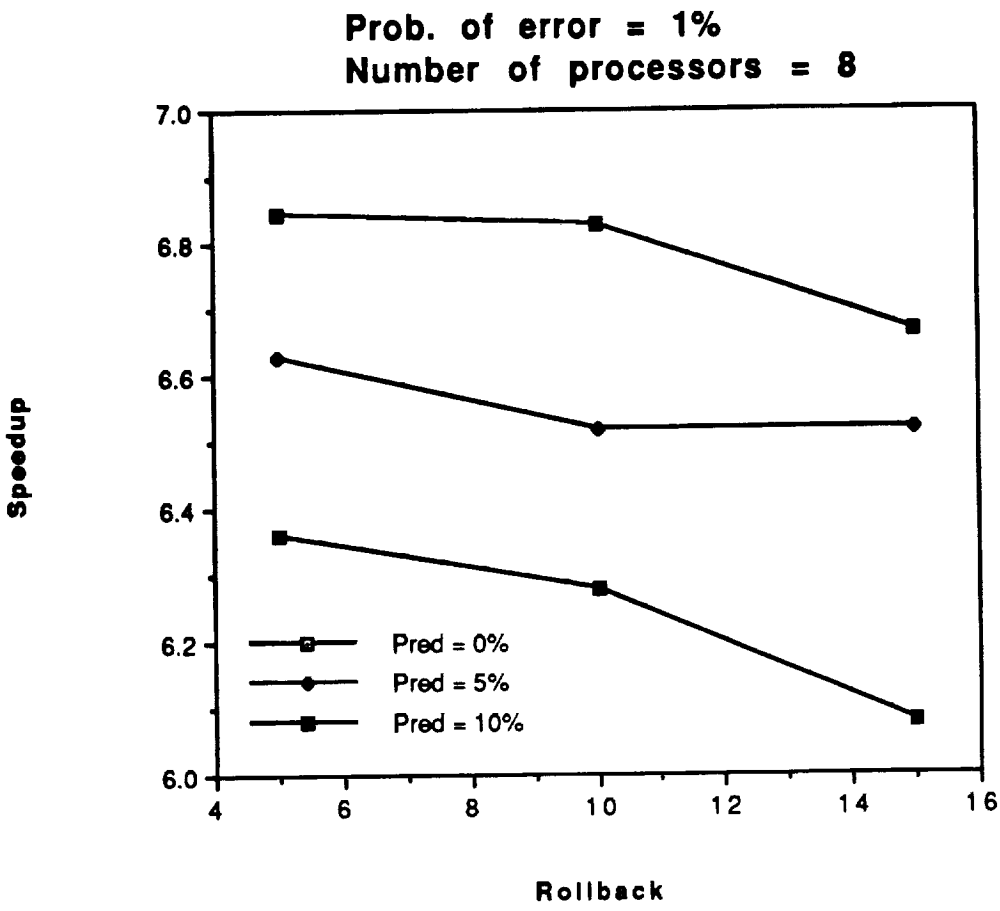

Rollback = 15%
Number of Processors = 8
Pred. = 0%
Pred. = 5%
Pred. = 10%
Speedup
7
6
5
4
0
10
20
30
40
Prob of error

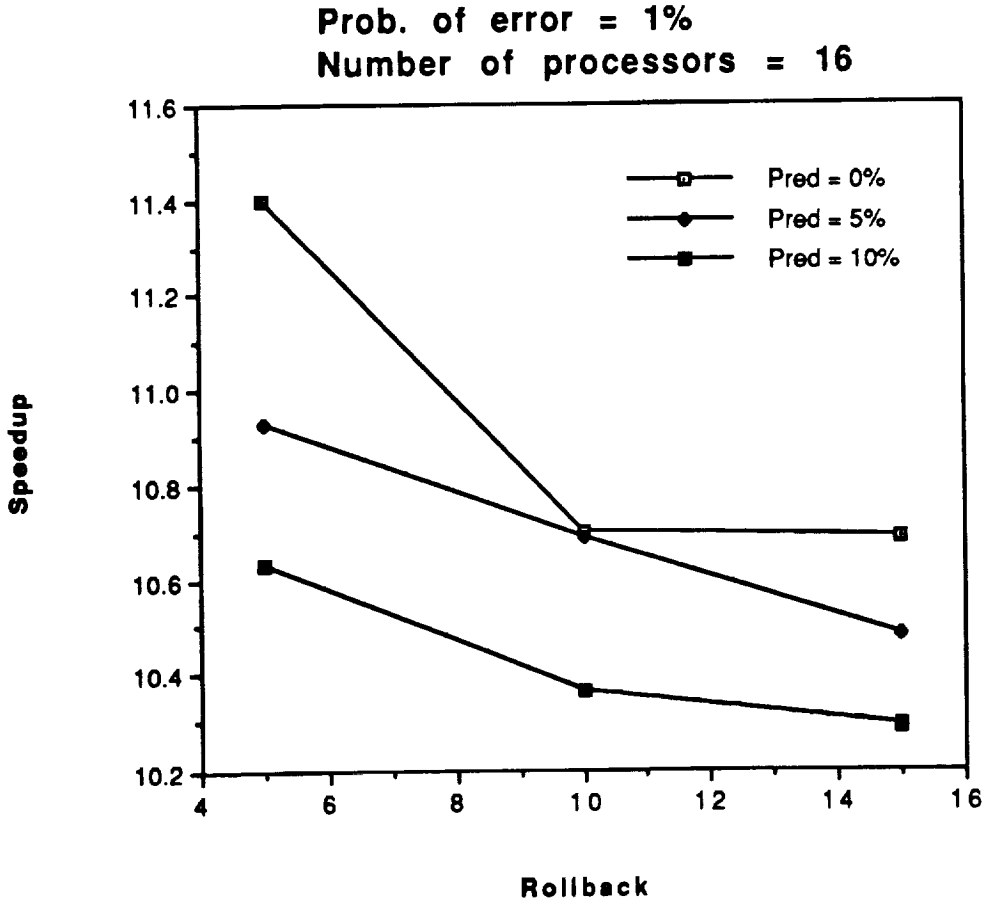

Rollback = 15%
Number of Processors = 16
Pred. = 0%
Pred. = 5%
Pred. = 10%
Speedup
11
10
9
8
0
10
20
30
40
Prob of error

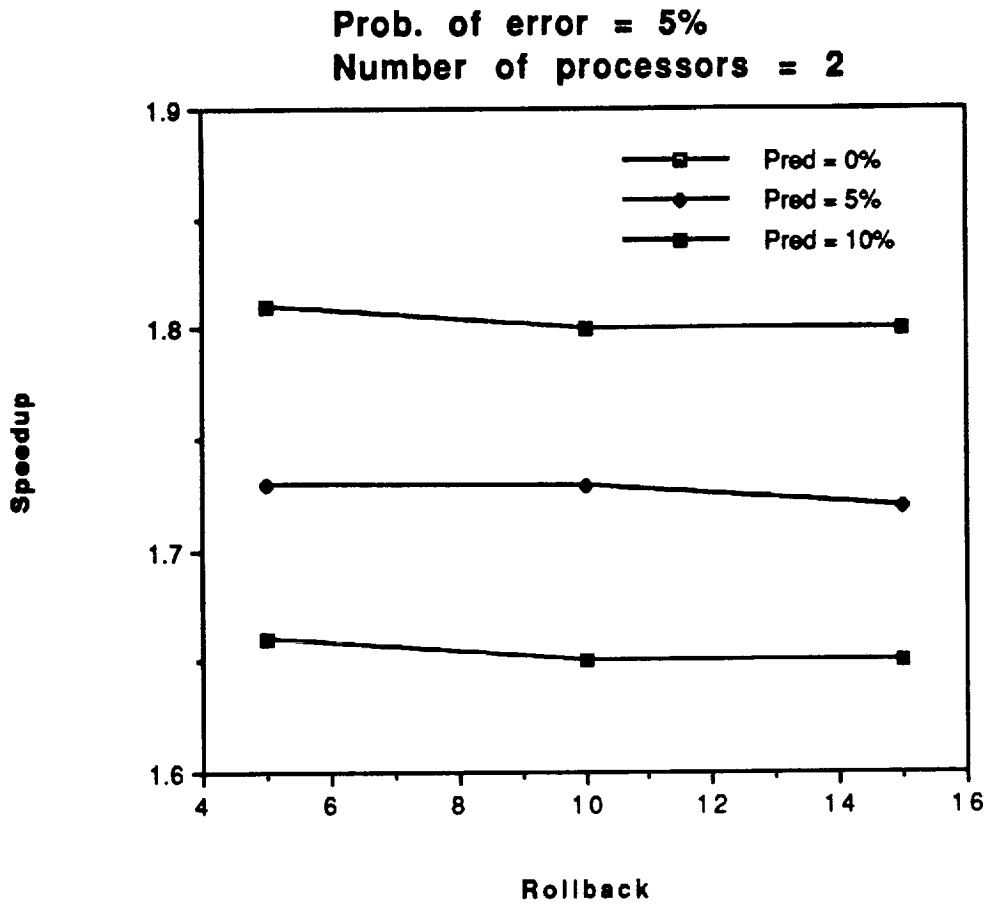
Appendix C

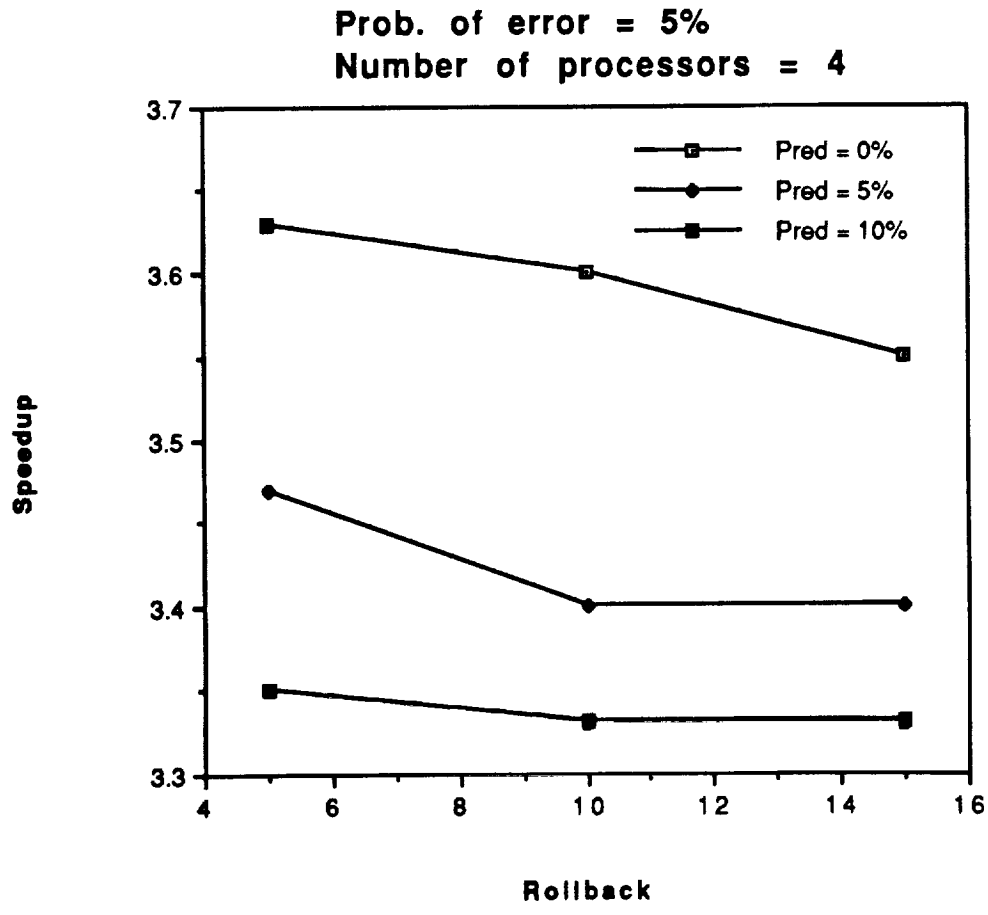

Prob. of error = 1%
Number of Processors = 2
Pred = 0%
Pred = 5%
Pred = 10%
2.0
1.9
1.8
1.7
4
6
8
10
12
14
16
Speedup
Rollback

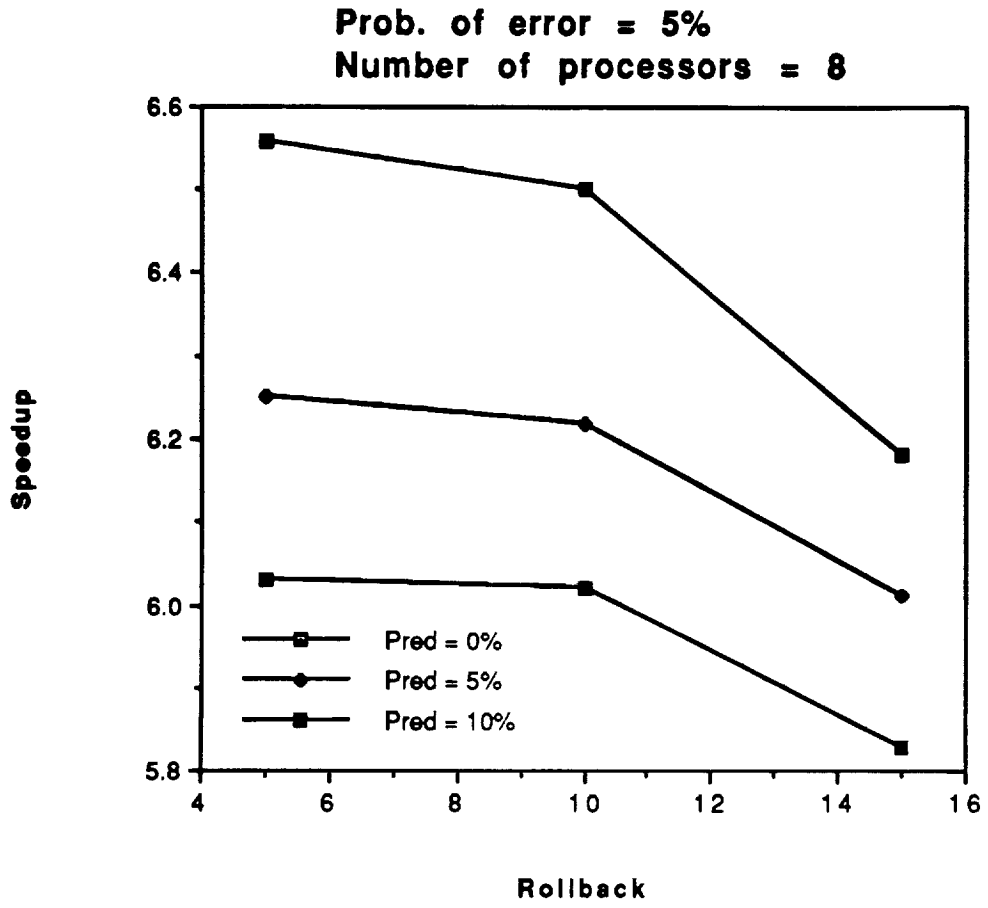

Prob. of error = 1%
Number of processors = 4
Pred = 0%
Pred = 5%
Pred = 10%
3.8
3.7
3.6
3.5
3.4
3.3
4
6
8
10
12
14
16
Speedup
Rollback

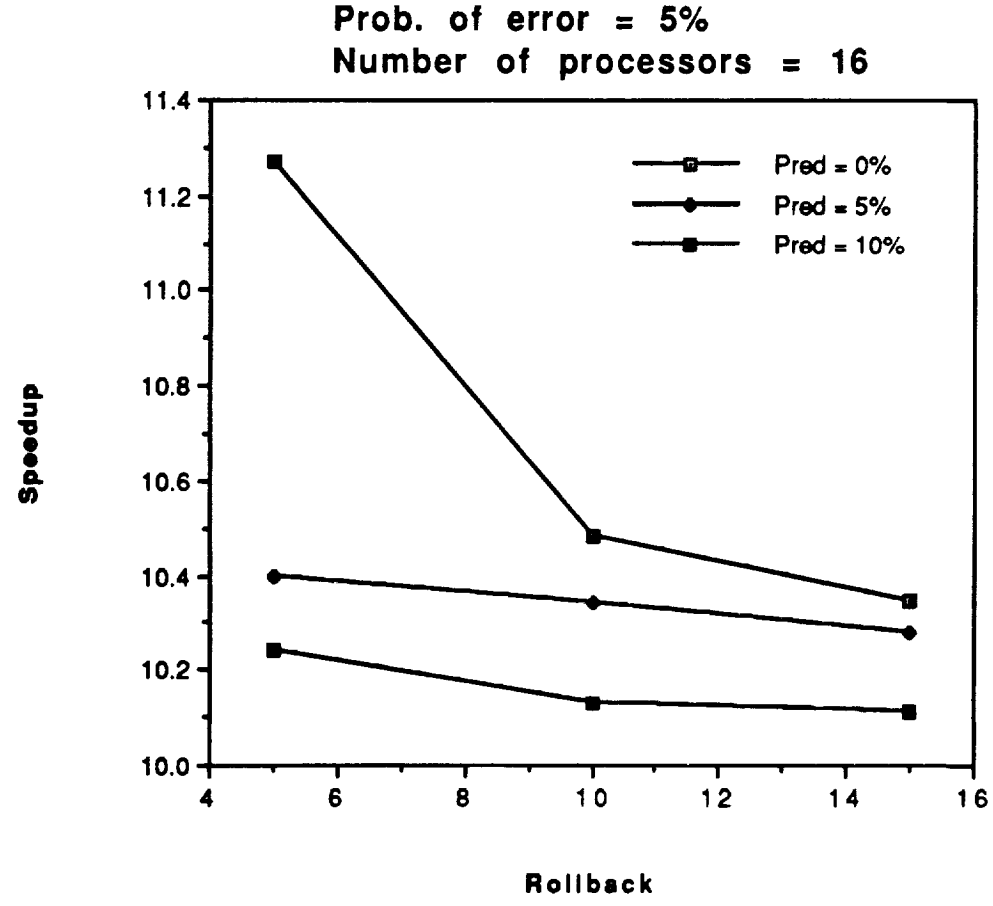

Prob. of error = 1%
Number of processors = 8
Speedup
7.0
6.8
6.6
6.4
6.2
6.0
4
6
8
10
12
14
16
Pred = 0%
Pred = 5%
Pred = 10%
Rollback

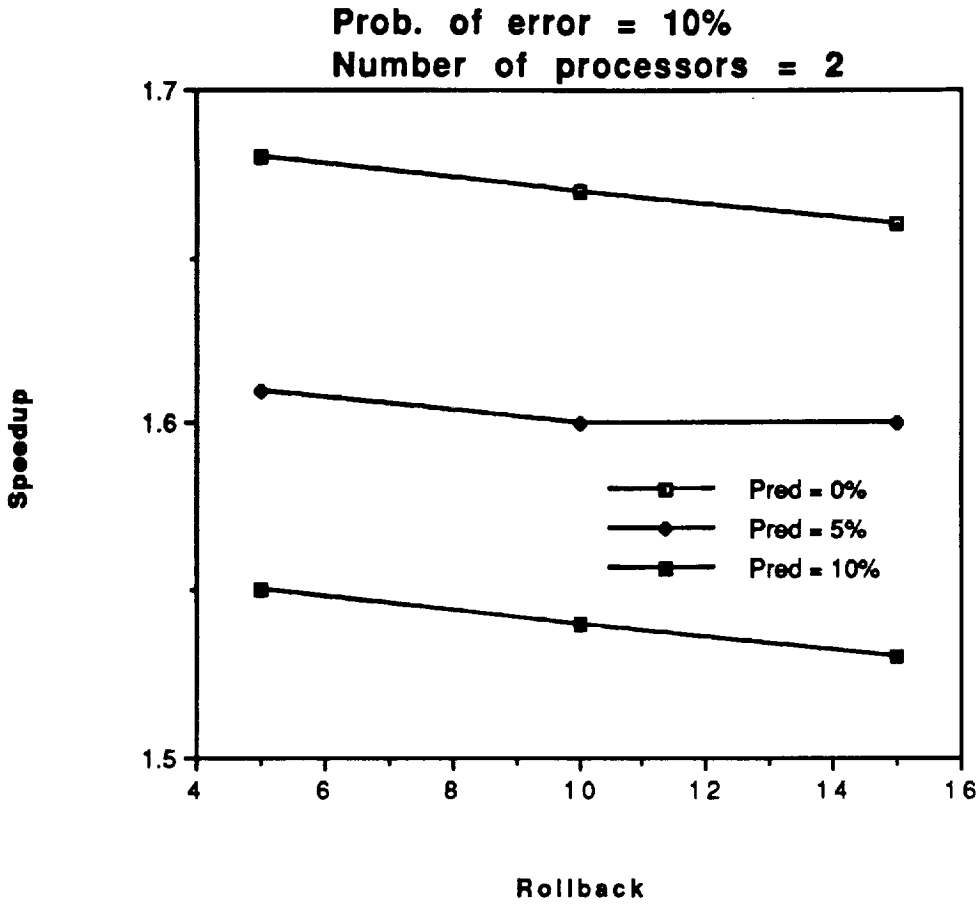

Prob. of error = 1%
Number of processors = 16
Speedup
11.6
11.4
11.2
11.0
10.8
10.6
10.4
10.2
4
6
8
10
12
14
16
Pred = 0%
Pred = 5%
Pred = 10%
Rollback

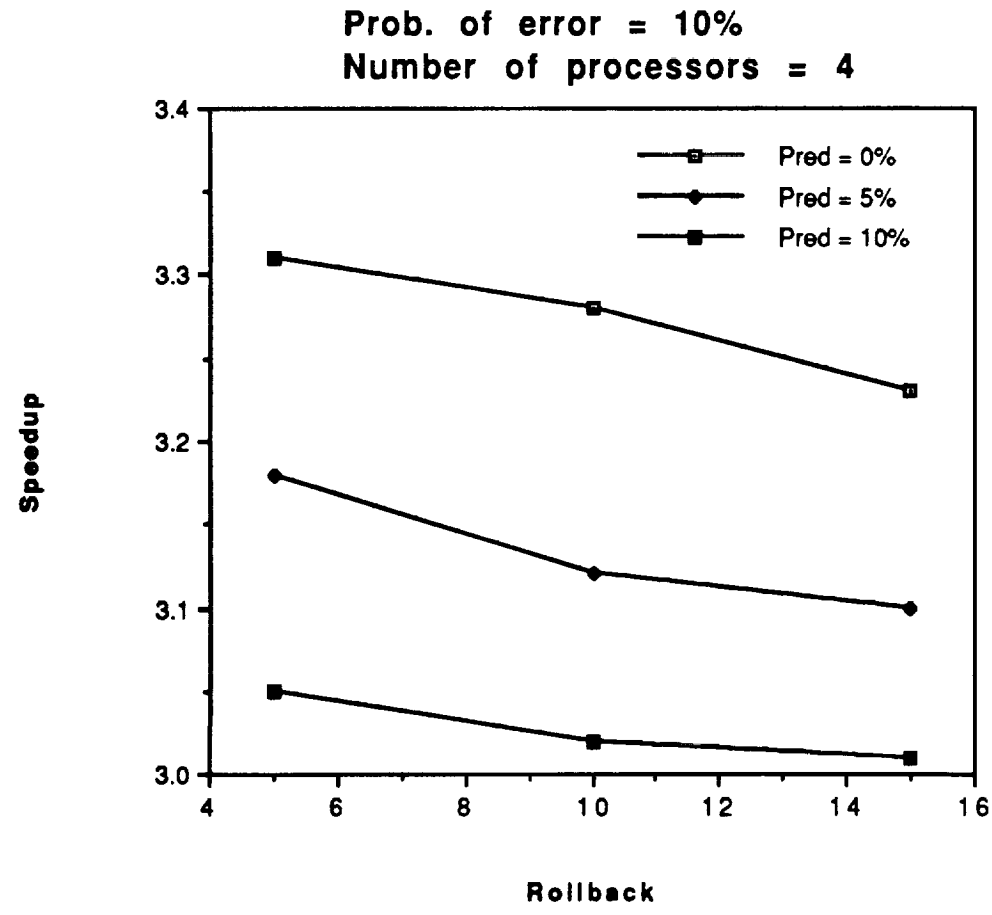

Prob. of error = 5%
Number of processors = 2
Pred = 0%
Pred = 5%
Pred = 10%
Speedup
1.9
1.8
1.7
1.6
4
6
8
10
12
14
16
Rollback

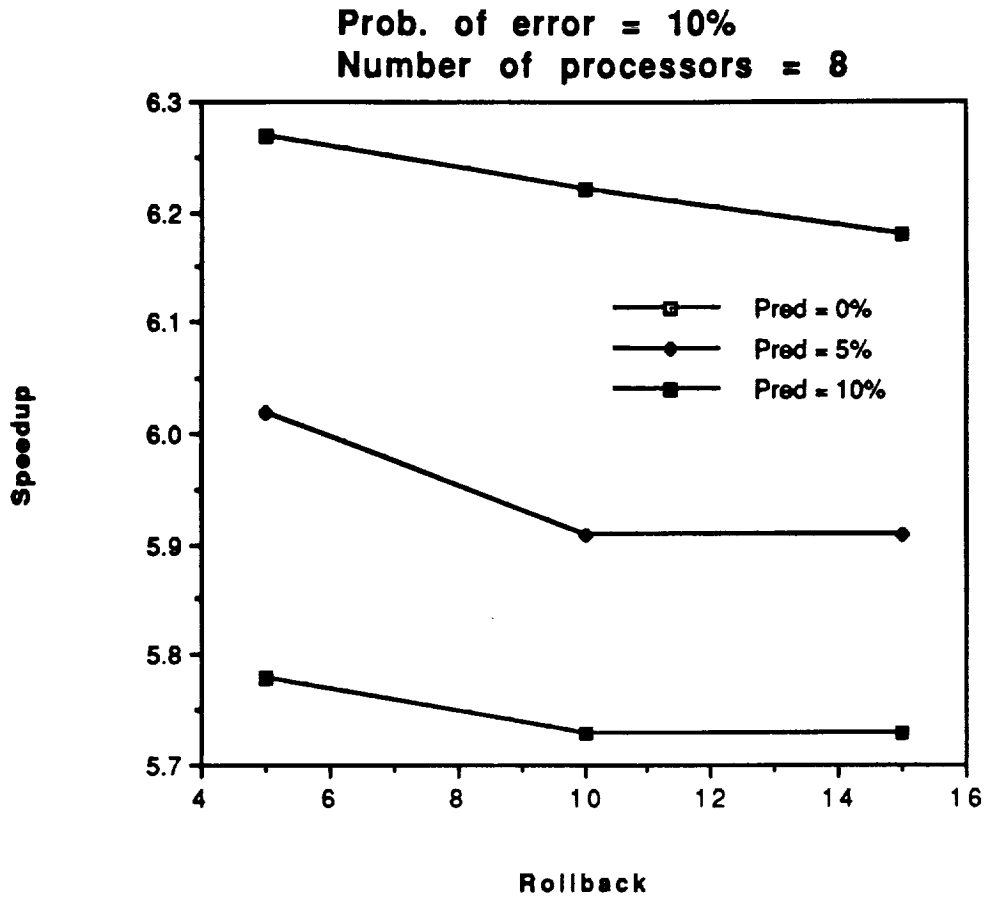

Prob. of error = 5%
Number of processors = 4
Pred = 0%
Pred = 5%
Pred = 10%
Speedup
3.7
3.6
3.5
3.4
3.3
4
6
8
10
12
14
16
Rollback

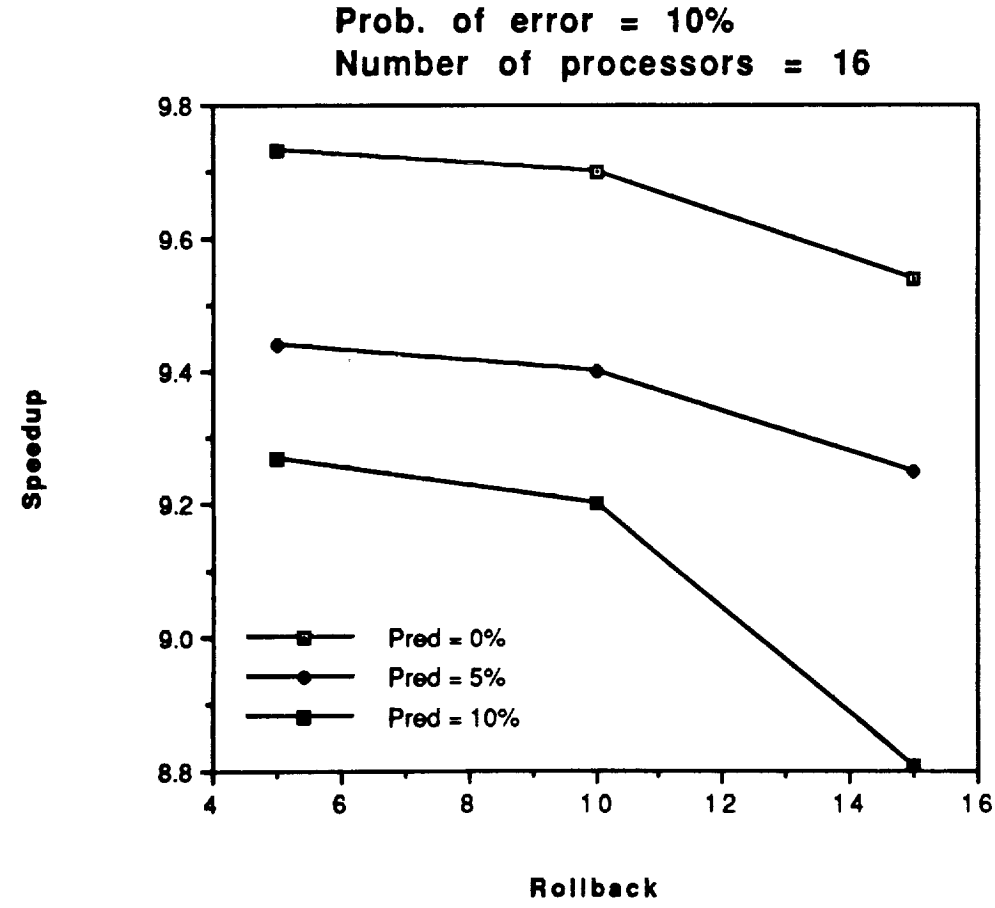

Prob. of error = 5%
Number of processors = 8
Speedup
6.6
6.4
6.2
6.0
5.8
4
6
8
10
12
14
16
Rollback
Pred = 0%
Pred = 5%
Pred = 10%


Prob. of error = 5%
Number of processors = 16
Speedup
11.4
11.2
11.0
10.8
10.6
10.4
10.2
10.0
4
6
8
10
12
14
16
Rollback
Pred = 0%
Pred = 5%
Pred = 10%


Prob. of error = 10%
Number of processors = 2
Speedup
1.7
1.6
1.5
4
6
8
10
12
14
16
Rollback
Pred = 0%
Pred = 5%
Pred = 10%


Prob. of error = 10%
Number of processors = 4
Speedup
3.4
3.3
3.2
3.1
3.0
4
6
8
10
12
14
16
Rollback
Pred = 0%
Pred = 5%
Pred = 10%


Prob. of error = 10%
Number of processors = 8
Speedup
6.3
6.2
6.1
6.0
5.9
5.8
5.7
4
6
8
10
12
14
16
Rollback
Pred = 0%
Pred = 5%
Pred = 10%


Prob. of error = 10%
Number of processors = 16
Speedup
9.8
9.6
9.4
9.2
9.0
8.8
4
6
8
10
12
14
16
Rollback
Pred = 0%
Pred = 5%
Pred = 10%

CPSC-311 Analysis of Algorithms

Fall 1992

Instructor: Dr. Jianer Chen

Office: ENSB 402B

Office Hours: MWF 10:30 - 11:30 am

Assignment # 2

(Due October 12)

1. Textbook, page 111, Problem 2.2.

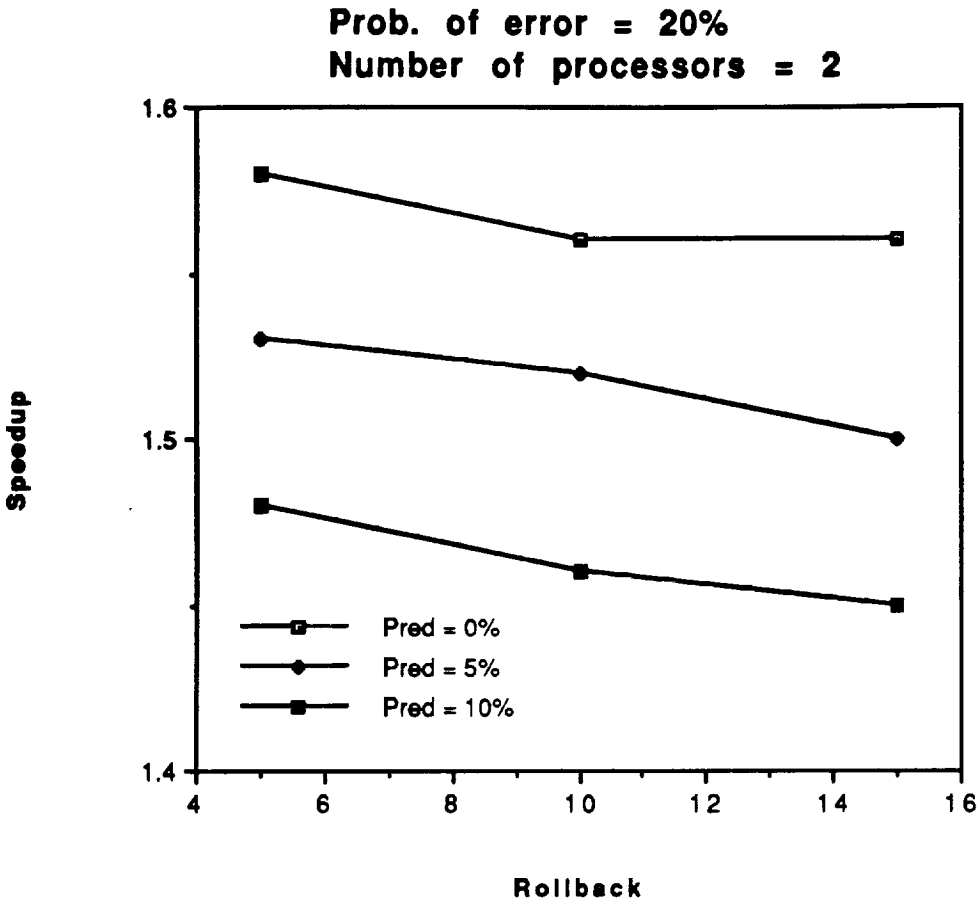
2. Textbook, page 115, Problems 2.16 and 2.20.

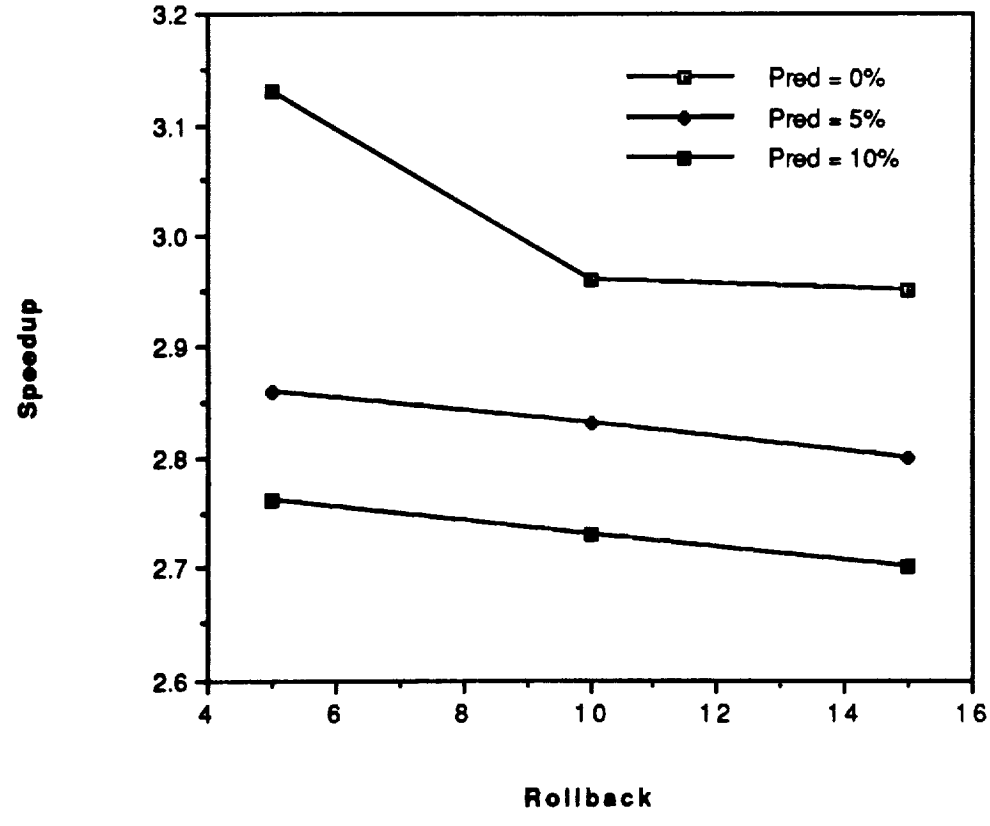
3. Textbook, page 116, Problem 2.26.

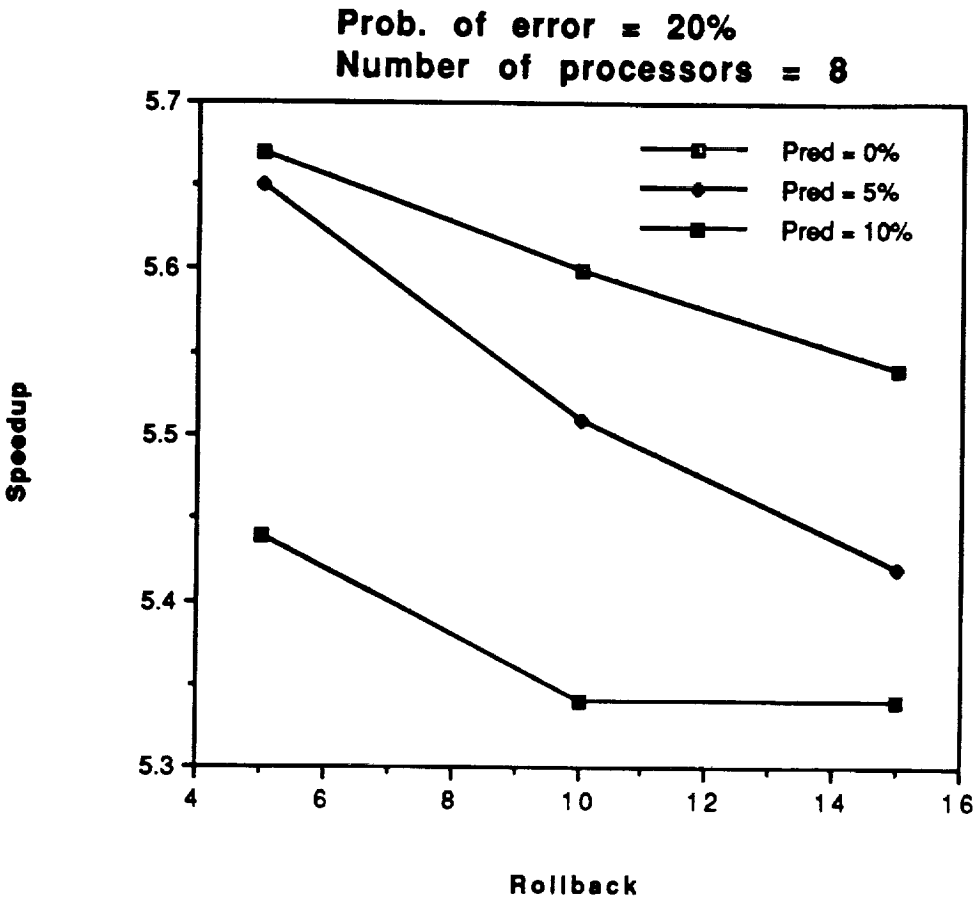
4. Textbook, page 117, Problem 2.34 a), c), d), e), g).

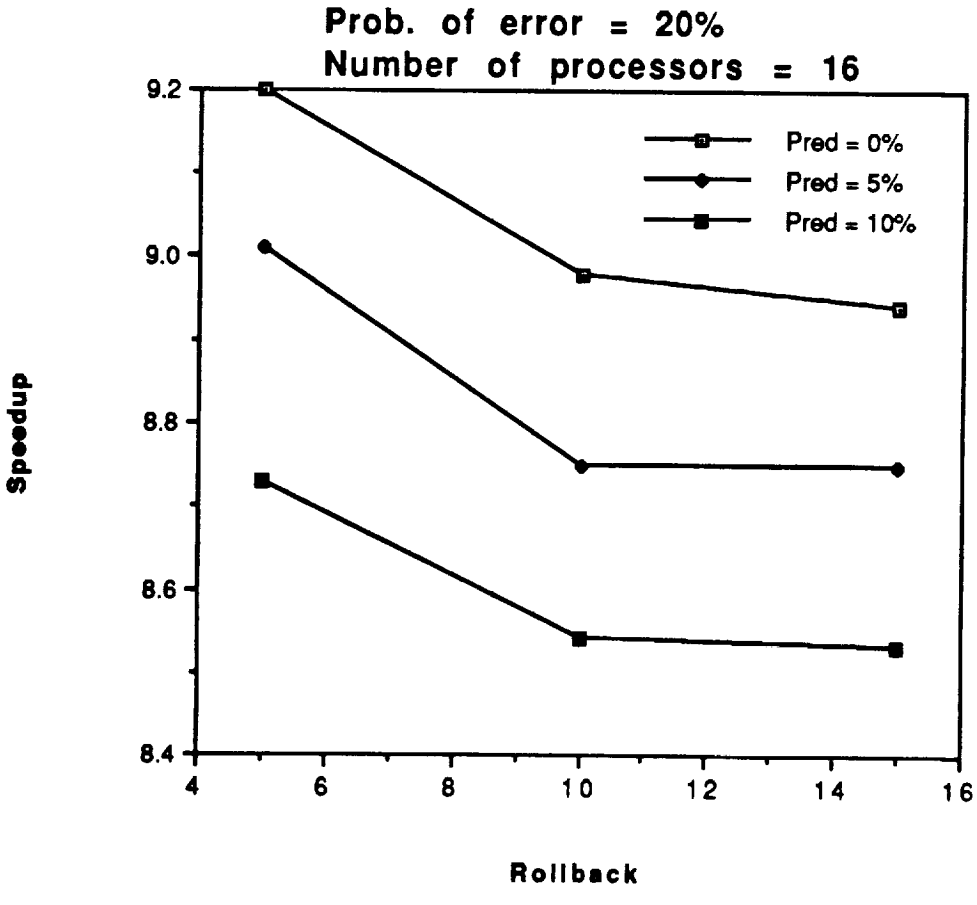
5. Textbook, page 141, Problem 3.12.

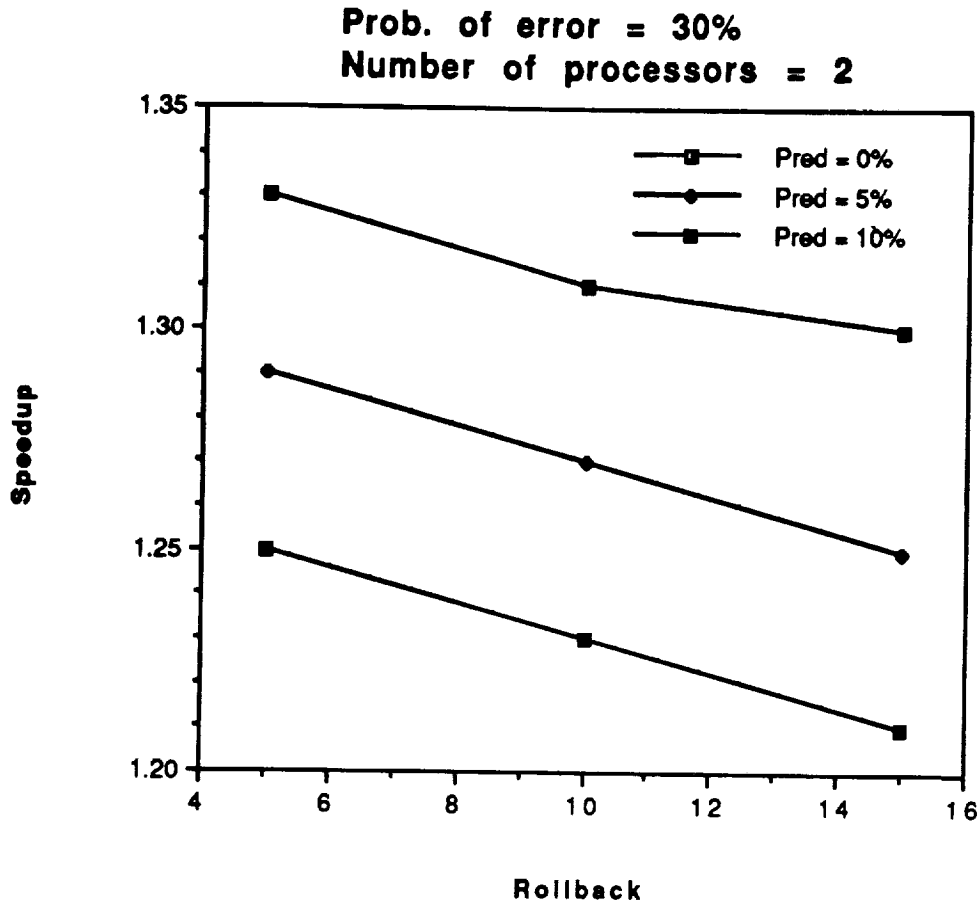
6. Textbook, page 142, Problem 3.19 a).

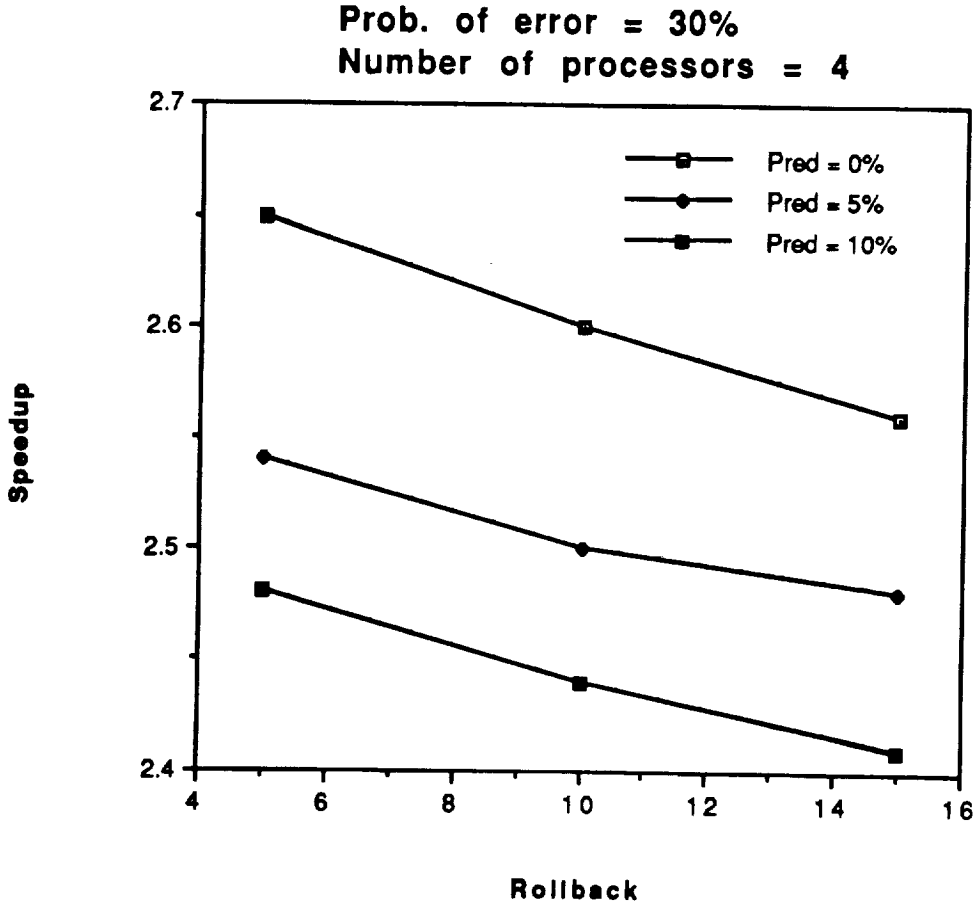

Prob. of error = 20%
Number of processors = 2
Speedup
1.6
1.5
1.4
4
6
8
10
12
14
16
Rollback
Pred = 0%
Pred = 5%
Pred = 10%

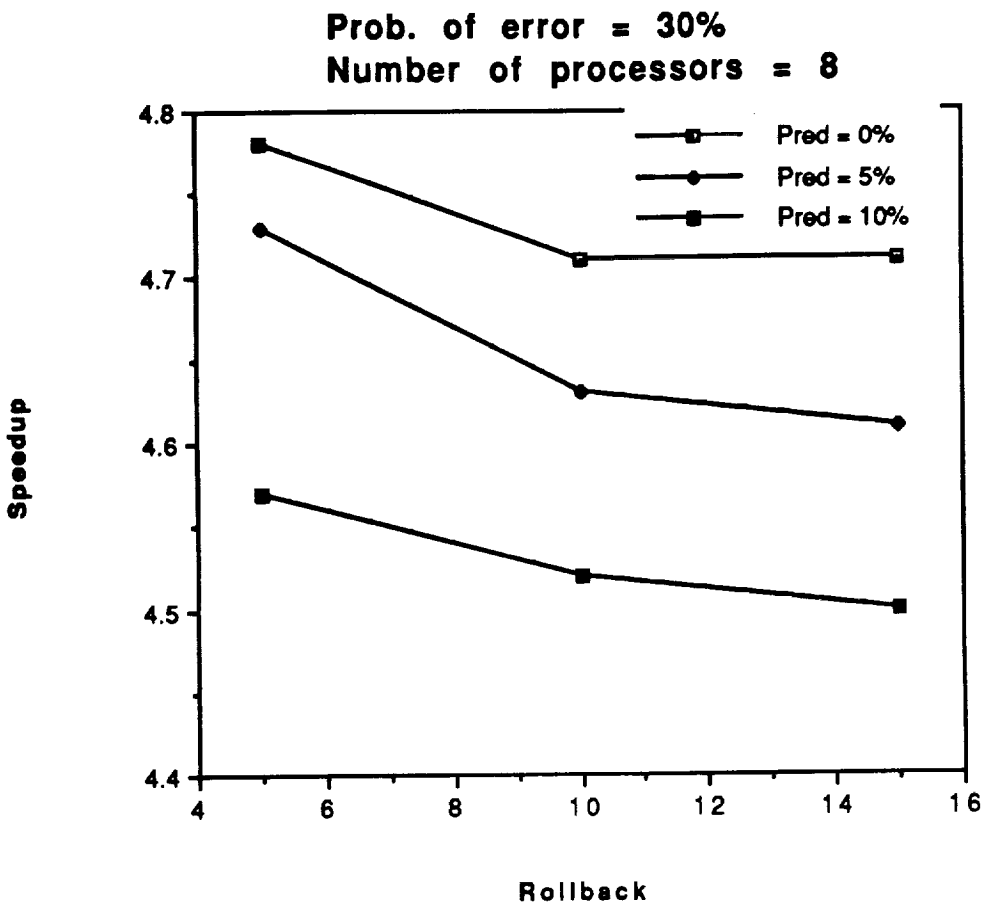

Prob. of error = 20%
Number of processors = 4
Speedup
3.2
3.1
3.0
2.9
2.8
2.7
2.6
4
6
8
10
12
14
16
Rollback
Pred = 0%
Pred = 5%
Pred = 10%

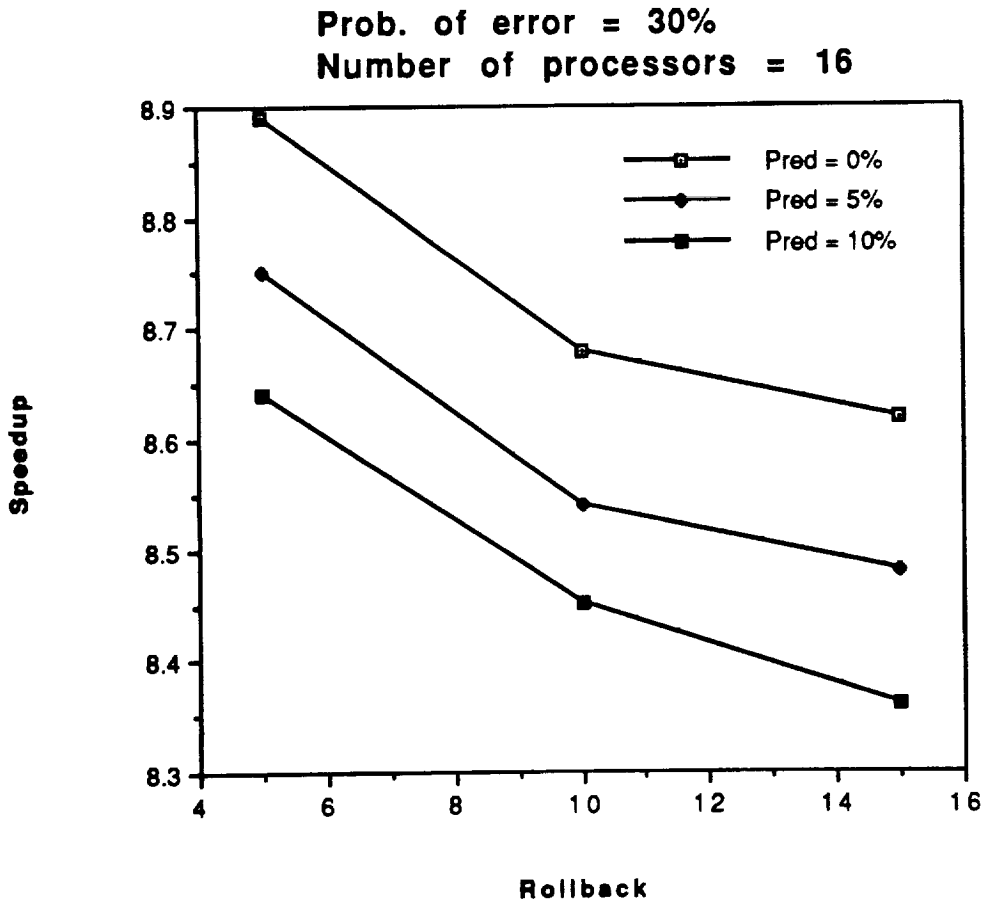

Prob. of error = 20%
Number of processors = 8
Pred = 0%
Pred = 5%
Pred = 10%
Speedup
5.7
5.6
5.5
5.4
5.3
4
6
8
10
12
14
16
Rollback


Prob. of error = 20%
Number of processors = 16
Pred = 0%
Pred = 5%
Pred = 10%
Speedup
9.2
9.0
8.8
8.6
8.4
4
6
8
10
12
14
16
Rollback


Prob. of error = 30%
Number of processors = 2
Pred = 0%
Pred = 5%
Pred = 10%
1.35
1.30
1.25
1.20
Speedup
4
6
8
10
12
14
16
Rollback


Prob. of error = 30%
Number of processors = 4
Pred = 0%
Pred = 5%
Pred = 10%
2.7
2.6
2.5
2.4
Speedup
4
6
8
10
12
14
16
Rollback


Prob. of error = 30%
Number of processors = 8
Pred = 0%
Pred = 5%
Pred = 10%
Speedup
4.8
4.7
4.6
4.5
4.4
4
6
8
10
12
14
16
Rollback


Prob. of error = 30%
Number of processors = 16
Pred = 0%
Pred = 5%
Pred = 10%
Speedup
8.9
8.8
8.7
8.6
8.5
8.4
8.3
4
6
8
10
12
14
16
Rollback